Verification of Pointer-Based Programs with Partial Information

LUO, CHENGUANG

Verification of Pointer-Based Programs with Partial Information

Chenguang Luo

A Thesis presented for the degree of
Doctor of Philosophy

School of Engineering and Computing Sciences
University of Durham
England

September, 2010

Dedicated to

Dad and Mom

for all their care and sacrifice

and also to

Shan

for all her love and support

Abstract

The proliferation of software across all aspects of people's life means that software failure can bring catastrophic result. It is therefore highly desirable to be able to develop software that is verified to meet its expected specification. This has also been identified as a key objective in one of the UK Grand Challenges (GC6) (Jones et al., 2006; Woodcock, 2006). However, many difficult problems still remain in achieving this objective, partially due to the wide use of (recursive) shared mutable data structures which are hard to keep track of statically in a precise and concise way.

This thesis aims at building a verification system for both memory safety and functional correctness of programs manipulating pointer-based data structures, which can deal with two scenarios where only partial information about the program is available. For instance the verifier may be supplied with only partial program specification, or with full specification but only part of the program code. For the first scenario, previous state-of-the-art works (Nguyen et al., 2007; Chin et al., 2007; Nguyen and Chin, 2008; Chin et al., 2010) generally require users to provide full specifications for each method of the program to be verified. Their approach seeks much intellectual effort from users, and meanwhile users are liable to make mistakes in writing such specifications. This thesis proposes a new approach to program verification that allows users to provide only partial specification to methods. Our approach will then refine the given annotation into a more complete specification by discovering missing constraints. The discovered constraints may involve both numerical and multiset properties that could be later confirmed or revised by users. Meanwhile, we further augment our approach by requiring only partial specification to be given for primary methods of a program. Specifications for loops and auxil-

iary methods can then be systematically discovered by our augmented mechanism, with the help of information propagated from the primary methods. This work is aimed at verifying beyond shape properties, with the eventual goal of analysing both memory safety and functional properties for pointer-based data structures. Initial experiments have confirmed that we can automatically refine partial specifications with non-trivial constraints, thus making it easier for users to handle specifications with richer properties.

For the second scenario, many programs contain invocations to unknown components and hence only part of the program code is available to the verifier. As previous works generally require the whole of program code be present, we target at the verification of memory safety and functional correctness of programs manipulating pointer-based data structures, where the program code is only partially available due to invocations to unknown components. Provided with a Hoare-style specification {Pre} prog {Post} where program prog contains calls to some unknown procedure unknown, we infer a specification $mspec_u$ for unknown from the calling contexts, such that the problem of verifying prog can be safely reduced to the problem of proving that the procedure unknown (once its code is available) meets the derived specification $mspec_u$. The expected specification $mspec_u$ is automatically calculated using an abduction-based shape analysis specifically designed for a combined abstract domain. We have implemented a system to validate the viability of our approach, with encouraging experimental results.

Declaration

The work in this thesis is based on research carried out at the Innovative Computing Group, School of Engineering and Computing Sciences, Durham University, England. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Acknowledgements

I would like to express my obligations to all the people who have helped me in the years for this PhD. First of all, I am extremely grateful to my supervisor Dr Shengchao Qin who initiated my research career and encouraged me to make an impact with my research. His wisdom enlightened me from the very beginning, his persistent support provided me with the spirit to overcome difficulties, and his thoughtful insights showed me the future directions to head for. Meanwhile his personal invitations (once and again) for me to join his family parties to enjoy my best Chinese food in the UK are also memorable.

I want to acknowledge my main colleague and office-mate Guanhua He. His knowledge over program analysis, his proficiency in functional programming, and his invaluable discussions on our research always benefited me. In addition, it is exceptionally amusing to be his neighbour for two years.

The work in this dissertation has been improved much through collaboration and discussion with my collaborators, Dr Florin Craciun and Prof Wei-Ngan Chin. Both are professionals in my area of research with great wisdom and rich experience. Dr Craciun inspired me on the work to deal with verification of programs invoking unknown components and helped me all the way through it. Meanwhile Prof Chin illuminated my work to refine program specifications with his expertise. I sincerely appreciate their help and support here.

Another research group that I want to bow to is the East London Massive, especially Dr Hongseok Yang and Prof Peter W. O'Hearn. I will always remember the times when I met Prof O'Hearn in Shanghai and when I saw Dr Yang in both Durham and York. Their continuous encouragement and precious discussions provided me with the confidence to continue with and improve our work.

When I just finished my first year of study, my progress examiners Prof Malcolm Munro and Dr Rafael Bordini confirmed my work and asked many questions which arouse my initiative to find my solutions to them. I am obliged to both of them, especially Prof Munro who is always well respected in our school.

I wish to present my gratitude to Prof Iain Stewart and Dr Hongseok Yang for accepting to join my thesis committee and examine my work. I appreciate their insightful comments and valuable suggestions to improve this thesis.

I am still urged to say thanks to all my fellow research associates/PhD students/MSc students in School of Engineering and Computing Sciences (in chronological order of my being acquainted with each of you) Yonghong Xiang, Qingzheng Zheng, Geng Sun, Wei Xiong, Fan Yang, Wei Lu, Yan Zhuang, Ying Yang, Rui Liu, Dingli Hu, Bindi Chen, Lu Chen and Hao Xia. It would have been a great pity had I missed any of you and it was always a great time to become an acquaintance of all of you. I will always remind me of the *auld lang syne* with you.

I acknowledge my *alma mater*, Durham University, for her provision of the Doctoral Fellowship. Without this essential support I would have never had the opportunity to study here as a happy PhD student.

Finally, I could not have completed this thesis without the love and support from my family. My parents have always been caring for me even from thousands of miles away in any forms that a person may or may not imagine. They also have made so many sacrifices for me to gain my current achievement (and I cannot express sufficient appreciation to them for any further trouble that I may cause to them as their son). My grandparents always asked me to video-call them to ensure that I was living happily such that they did not have to worry. Also my brothers and sisters frequently imparted their latest news to me to cheer me up. And last but not least, I must acknowledge my wife (including her family), for her everlasting companion and deep love. All my effort would have been in vain had her never appeared in (the toughest time of) my life, and I could not possibly thank her enough (but I still have to) because of her courage and determination to share our lives with each other.

Contents

List of Figures

List of Tables

List of Tables

Chapter 1

Introduction

Since the invention of computers in the last century, computer-based systems are playing an increasingly more significant role across all aspects of people's lives. Accordingly, software systems behind these computer systems are being widely used as their souls.

Such proliferation of software means that software failure can bring catastrophic result. The worst aftermath of software failure could be that, under safety-critical circumstances, it can put human lives in danger. Two extreme examples include the Patriot missile system failure resulting in 28 US soldiers killed and another 98 injured, which was caused by float rounding problem (Information Management and Technology Division, 1992), and the Therac-25 radiation therapy machine having overdosed at least six people with three deaths, attributed to its race hazard (Leveson and Turner, 1993). Currently such safety-critical systems are even more widespread like the embedded systems in airplanes and automobile vehicles, as well as contemporary medical devices, any of which is highly responsible for human lives.

What is more, misbehaviour of software also causes great economical loss. Some notorious cases follow the Ariane-5 explosion due to an overflow error in the conversion from 64-bit float to 16-bit integer which cost over 370 million US dollars (Lions, 1996), and the loss of another 125 million US dollars by a Mars Orbiter crash resulted from improper usage of Imperial units and metric units (Stephenson et al., 1999). Other less severe examples can be witnessed by the "blue screens" of unexpected system halt from Microsoft Windows as well as "segmentation faults" from POSIX-oriented operating systems, which cause frustration and loss of productivity. According to Research Triangle Institute (2002), the annual cost incurred by inadequate software quality in the United States is between 22.2 billion and 59.5 billion US dollars, which corresponds to approximately 0.2 to 0.6 percent of the country's gross domestic product (GDP).

Nowadays, to satisfy the fast development of demand from all aspects of the society, computer software is still growing in both scale and complexity, and its quality assurance draws increasing number of eyeballs accordingly. This has also been identified as a key objective in one of the UK Grand Challenges (GC6) (Jones et al., 2006; Woodcock, 2006). Cousot has ever addressed this point as "it is preferable to verify that mission-critical or safety-critical software programs do not go wrong before running them" (Cousot and Cousot, 2010). Hence people are trying to search for innovative ways such as verification and analysis leading to software that has better quality and thus is more dependable.

1.1 A Brief Overview of Approaches to Software Quality Assurance

There are various approaches to the reduction of software bugs and the improvement of software quality. A brief (and not exhaustive) overview includes programming language design, software testing, model checking, program verification, program analysis, and so forth.

To eliminate certain kinds of programming errors, the design of suitable programming languages that totally prevent them may sound a nice idea (Jim et al., 2002; Condit et al., 2007). A case in point is the programming languages with garbage collectors, such as Java (Venners, 1999), which claim to free programmers from heap space exhaustion caused by forgetting to deallocate heap objects. However, this approach could not solve the class of errors that already exist in current software or software not developed with such language (say C programs). Meanwhile, the design of these languages may introduce extra complications, for instance the garbage collector thread in Java virtual machine will block any other threads and thus making it not very suitable for some real-time systems (Petit-Bianco, 1998).

Another significant approach to software quality guarantee is testing. It is an investigation conducted to provide people with information about the quality of the product or service (software) under test (Kaner, 2006). Currently it is the most prevalent approach employed by people, by setting out certain input for the program to execute and observing whether or not it proceeds as expected. For example, for a piece of code $\mathtt{x} = \mathtt{x}/(\mathtt{y}+\mathtt{1})$, when supplied with a test-case $\mathtt{x} = \mathtt{0}, \mathtt{y} = -\mathtt{1}$ we will discover a bug of unprotected division by zero. As can be seen, this approach is relatively easy and straightforward to be applied in the process of software development, as essentially it only requires to run the program (although there are also

many investigations and tools to assist such process (Hetzel and Hetzel, 1991; Craig and Jaskiel, 2002)). Yet one of its most severe defect is that it is not exhaustive, and hence cannot prove the absence of bugs (as in many cases we cannot exhaust all inputs to a program for testing). There are methods to alleviate this problem, such as systematic testing (Godefroid, 1997; Musuvathi et al., 2002) to design test-cases covering more (or even all of) program execution paths, but it still does not provide a bug-free proof for *every* input.

Compared with the "dynamic" method of testing, a number of "static" approaches attract more and more attention, such as software model checking, program verification and program analysis. These approaches do not require running the program; instead they analyse the code in some way to prove that the program satisfies certain properties (hence they are categorised as "static") (Nielson et al., 2005). Unlike testing, these properties usually exhaust all possible inputs to the program (say "deadlock-free" or "termination" under all circumstances), so they guarantee that the program is free of bugs with respect to these properties. The expense to achieve such merits is that some specific techniques to transform/analyse the program are necessary, such as abstraction (to avoid undecidability). Meanwhile these methods are generally quite costly in terms of time/space complexity due to the static reasoning performed over the sophisticated software which they check/verify/analyse.

Of all these three approaches, model checking was originally developed to verify finite-state systems (such as the design of circuits) by exhausting the whole set of computation states according to some specifications (McMillan, 1992). Later it was extended to the field of software quality assurance with the help of abstraction techniques to reduce the scale of possible software states (Ball et al., 2001). In this thesis we would rather focus on the other two approaches, namely, program verification and program analysis. Program verification uses a deductive reasoning system to verify whether a program conforms to user-supplied specifications (Floyd,

1967; Hoare, 1969). These specifications contain additional information that expresses developer's designs and other application-specific properties, which allows program verifiers to generate logical formulae whose validity entails the consistency of the program. For instance, a verifier will accept as consistent a piece of code with annotation:

$$\{\texttt{true}\}\ \texttt{if}\ (\texttt{x} > \texttt{y})\ \texttt{z} = \texttt{x}\ \texttt{else}\ \texttt{z} = \texttt{y}\ \{(\texttt{x} > \texttt{y} \wedge \texttt{z} = \texttt{x}) \vee (\texttt{x} \leq \texttt{y} \wedge \texttt{z} = \texttt{y})\}$$

which addresses that whatever state the if-statement starts with (`true`), `z` will be assigned as the larger value of `x` and `y`. Note that the specifications cover infinite concrete variable values, which is distinct from testing. One problem that verification has resides with the annotations: they must be supplied by programmers, which work is both tedious and error-prone. Therefore program analysis is born to solve this problem, by analysing the program code and *inferring* the specifications that the program should conform to, instead of asking the users to provide them (Nielson et al., 2005). To illustrate, for the previous example it may analyse the bare code to find out the specifications in the curly brackets. This eases much of the programmers' work; however, due to the complexity of programs, the analysis could be very expensive for some interesting program properties. This trade-off partially motivates the work presented in this thesis, as will be described in the next section.

1.2 Motivation

As aforesaid, it is highly desirable to develop software with assured quality, and one feasible approach to achieving this objective is program verification and analysis. However, although research on this aspect has a long and distinguished history since Floyd (1967); Hoare (1969), it remains a challenging problem to automatically verify programs written in mainstream imperative languages such as C, C++, C♯ and Java. This is in part due to the wide use of (recursive) shared mutable data

structures allocated in heap memory, for example singly or doubly linked lists, binary trees, and their variants like sorted lists and binary search trees. Compared with previous concentration of program verification on simpler properties as valuation of variables (say $\mathtt{x} \neq \mathtt{0}$ and $\mathtt{y} \geq \mathtt{x} + \mathtt{1}$), these data structures are much more difficult to keep track of in a precise and concise way for program verification/analysis.

The emergence of separation logic (Ishtiaq and O'Hearn, 2001; Reynolds, 2002) brings dramatic advances in automated verification and analysis of such heap-manipulating programs. As an extension of Hoare logic (Hoare, 1969), it presents a framework to reason about these programs by modelling the program memory (both stack and heap) in a natural and accurate manner. Therefore, on the basis of separation logic, many works witness the progress of verifying the heap-manipulating programs, such as the Smallfoot tool (Berdine et al., 2005b) for the verification on lists' pointer safety (i.e. program properties asserting that pointers cannot go wrong), the verification on termination (Berdine et al., 2006), the verification for object-oriented programs (Chin et al., 2008; Parkinson and Bierman, 2008), and Dafny (Leino, 2010), and HIP/SLEEK (Nguyen et al., 2007; Chin et al., 2007; Nguyen and Chin, 2008; Chin et al., 2010) for more general properties (including both *shape* information like linked lists and trees, and relevant quantitative information like list length and sortedness, and tree height and binary search property, etc). Here (and throughout the thesis) we use "shape" information/properties to denote the types of components of data structures located in memory and the spatial relationship among these components, for example "pointer `x` points to the head of a singly linked list, which is disjoint from a binary tree whose root is referenced by pointer `y`", and so on.

On the analysis side, the SpaceInvader tool (Distefano et al., 2006; Yang et al., 2008; Calcagno et al., 2009) automatically infers method specifications and loop invariants on pointer safety for list-manipulating programs. Other works such as

THOR (Magill et al., 2007, 2008, 2010) incorporate simple numerical information to allow automated synthesis of properties like list length. Their success proves the necessity and feasibility for program analysis to help automate the verification process.

Among these works, one state-of-the-art verification system is Hip/Sleek (Nguyen et al., 2007; Chin et al., 2007; Nguyen and Chin, 2008; Chin et al., 2010). Its capability covers the verification of memory safety, such as lists' pointer safety properties handled by Smallfoot and SpaceInvader, and some relatively simple numerical properties (also of lists) handled by THOR. Moreover, it also targets at both memory safety and functional correctness of programs. To achieve this objective, it allows users to specify their preferred level of program correctness by defining predicates to depict the data structures employed in their programs. Users may describe the spatial relationship amongst components of their data structures, or their quantitative features like size/height/sortedness, or even collections of values stored in the data structures. Reasoning about these predicates enables Hip/Sleek to verify both memory safety and functional correctness of heap-manipulating programs.

Besides its power and benefit, Hip/Sleek can still be improved in many aspects. One observation is that, same as many other existing verification systems, it requires ample information for verification, say the whole of program code and corresponding specifications. Two accompanying questions are: what if (part of) such information is not available? If it is unavailable, can we still discover such missing part of information to complete the verification? This forms the motivation of this thesis.

A first incentive to improve Hip/Sleek dwells in its requirement of user-supplied annotations. Such annotations include the specification for each method and the invariant for each loop in programs. Hip/Sleek's reliance on users to provide these annotations severely increases users' workload and can be a source of handmade

errors. This situation would be made better if some program analysis techniques could be exploited in such verification systems to reduce human labour by enabling them to write fewer such annotations.

Another motivation of my thesis is based on the requirement of Hip/Sleek to have access to the whole code being verified, which is usually not satisfied in contemporary programs making use of many third-party codes and/or libraries. In this case, part of the program code to be verified could be unknown to the verifier, and Hip/Sleek (as well as many other similar verification systems) does not have any solution but can only report "some procedure invoked in the program is not defined". Therefore, it should be quite useful if we enhance Hip/Sleek so that it can deal with such partially available programs with unknown components.

1.3 Objectives

The main objective of my thesis is to improve Hip/Sleek-like verifiers for heap-manipulating programs such that they can deal with two scenarios where only partial information about the program code is available. In one of the scenarios, users are just expected to provide partial specifications of the program, and thus their workload is reduced; in the other case the program may contain invocations to unknown components and hence only part of the program is available to the verifier. The success of such an effort would pave the way for a more powerful software verifier capable of handling the above two cases. More specifically, we want to achieve the following goals to enable our verifier to accept the following:

- **Partial specifications.** Original Hip/Sleek requires full specifications of programs to verify them. Our first objective is to allow users to provide only

shape information about data structures but not other information on numerical or content part, in order that their workload is reduced.

- **Partially available programs with unknown components.** When a program contains unknown components, we may still want to verify it, by finding some extra obligations that the unknown part of the program should conform to.

Meanwhile, during the meantime of realising these objectives, we will target at verifying both memory safety properties and functional correctness properties, by providing users with the flexibility to specify different kinds of data structure invariants and correctness properties in the same framework, which can then be verified using the same machinery. This is in contrast to specialised approaches, such as Rugina (2004); Habermehl et al. (2010), which are designed to work with restricted, "built-in" data structures.

1.4 Challenges

We face several challenges when working on this thesis. These challenges correspond to the two major objectives: verification automation by inferring missing information from the given program and partial specifications, and verification for programs with unknown components.

- **Verification automation.** The challenge presented by verification automation is mainly attributed to the inference of missing constraints to complete the given partial specifications. Such completion may require certain form of fixed-point calculation in the presence of loops/recursive calls over a combined

domain, as our verification generally works with both structural properties and other relevant numerical/content properties. Details of difficulties are listed as follows:

- **Incomplete information during verification.** Because user-provided specifications may be incomplete, both preconditions and postconditions can be too weak due to missing information. This has two consequences: a too weak precondition may lead to infeasible abstract program states during verification (as it might have missed some information necessary for memory safety), while a too weak postcondition is neither sound for recursive invocation nor satisfactory in terms of precision. Therefore we need some techniques to recover information at both sides.
- **Combination of various domains/theories.** As introduced, we expect programs to operate on different domains and data structures and we are able to handle many of these properties over multiple domains. For example, one single program can handle both lists and numbers at the same time, and our verification should fully capture all its behaviour (instead of only some behaviour over shapes). Therefore we have to combine provers and decision procedures for a range of theories that we work on. Existing methods, such as Nelson and Oppen (1979), often put restrictions that significantly affect the expressivity of the constituent theories. When we reason about structural and quantitative aspects of data structures at the same time, we definitely need some way to combine theories.
- **Abstraction.** Reasoning about programs in their full details is impractical. Thus we utilise abstractions to filter out irrelevant details so as to keep the reasoning within the reach of automated tools. Moreover, abstraction can improve efficiency as well. Choosing the right abstraction that keeps as much of relevant details and drops as much of irrelevant de-

tails is a non-trivial problem. Furthermore, we may also need abstraction to handle infinite structures in programs, such as recursive data structures, loops and recursion for termination purposes.

- **Aliasing.** Aliasing has long been an elusive problem for program verifiers. Aliases manifest themselves in many different forms (Bornat, 2000): variable aliasing, parameter aliasing, etc. The major difficulty caused by aliases is that updates to one component may affect other seemingly unrelated components.

- **Verification for programs with unknown components.** In this part, the main challenge is the part of program that is not available, which causes unknown state of the program during verification. This is detailed in the aspects below that we need to sidestep:

  - **Unknown components.** The unknown components in the partially available programs prevent the verification process from obtaining their code and therefore behaviour (as we do not have any knowledge about those components), so we cannot progress with the remainder of the program. We need to search for possible techniques to circumvent them before continuing with the verification.

  - **Verification framework.** As aforementioned, a program verifier should be a deductive reasoning system over program code and specifications. However, in the scenario where some code is missing, we must revise slightly the whole framework of verification to cater for it. This results in a verification framework based on both traditional program reasoning and *abductive* reasoning, which will be portrayed in more detail in Chapter 6.

  - **Combination of various domains/theories.** This issue plus the unknown components presents a new challenge for us, as we also work with a combined domain to express multiple sorts of program properties.

1.5 Evaluation Criteria

The goal of this thesis is to develop mechanisms for verification of heap-manipulating programs with respect to both memory safety and functional correctness, based on only partial information (either specifications or program code). Accordingly we set the evaluation criteria for this thesis in two aspects, viz., successful program verification with only partial specifications or only partial program code, respectively. This is detailed as follows:

- **One objective is to allow users to provide only partial specifications for verification, in order to reduce the amount of annotations provided by users.** With the thesis' work, users should only provide shape information about data structures but not the information on numerical/content part. Our proposed verifier will take over the rest of the work to *refine* the specification by discovering the missing part to make the specification become sound with respect to the program being verified.

- **The other objective is to verify partially available programs with unknown components.** When the program code is only partially available because of the existence of unknown components, we should still be able to verify the program in some way, by discovering an obligation for the unknown components to satisfy in order that the whole program is verified as correct.

- **Both objectives should be fulfilled with proper implementation and experiments to prove their feasibility.** The targeted experimental programs are programs implementing classical algorithms for pointer-based data structures, which manipulate the heap memory with subtle operations, resulting in rich specifications expressing shape, numerical and content properties of the algorithms.

1.6 Organisation of the Thesis

The current thesis comprises 8 chapters including this introductory chapter. The remainder of the thesis is organised as follows.

Chapter 2 presents a survey of the state-of-the-art works that are relevant to my work. It mainly introduces contemporary program verification and program analysis techniques, as well as separation logic as the foundation of my work to model abstract program memory states.

Chapter 3 presents the programming language we aim to verify, together with its operational semantics for soundness issue. The specification language describing the abstract domain will also be introduced with examples and semantics.

Chapter 4 presents the first contribution of this thesis, namely, the refinement of partial specifications for verification. It illustrates the main approach with some examples and formalises the pure abduction mechanism and the program analysis for verification purpose thereafter.

Chapter 5 presents the thesis' second contribution: the synthesis of specifications for auxiliary methods. It first illustrates how the users' workload can be further cut down for the example in the previous chapter, followed by the formalisation details.

Chapter 6 presents the third contribution of the thesis, viz. the verification of programs with unknown components. In a similar fashion as previous chapters, there is a motivating example to illuminate the approach, before the formal mechanism of enhanced abduction and framework of verification are gathered.

Chapter 7 presents our system implementation, the experimental results, and our observations and experience with the evaluation of the proposed approaches.

Chapter 8 concludes the thesis and discusses possible future works.

1.7 Summary

This chapter has presented an overall view of this thesis. It introduces briefly the current approaches to software quality assurance, sets out the motivation and objectives of this thesis, describes the challenges faced and the criteria to evaluate the work accomplished by the whole thesis. Finally it exhibits the organisation of the remainder of this thesis.

Chapter 2

Literature Review

The general idea of program verification and analysis has a long and distinguished history in computer science. As early as in the 1960s and 1970s, provident people have already foreseen the potential need of approaches to formal reasoning of increasingly sophisticated computer programs (Floyd, 1967; Hoare, 1969; Dijkstra, 1976). After having developed for over 40 years, this field has witnessed many significant techniques (such as modelling, verification and analysis) grown up, and it always encourages prominent researchers to pay their effort and form an active community of research. Upon their achievements, this chapter will provide a brief survey from the initialisation to the state-of-the-art in this area, so as to locate the position of my thesis among them.

As we target at the verification of pointer-based programs with partial information, the survey presented here mainly comprises relevant research of this work. Its main content is divided into two parts. The first part is about some techniques exploited in our work, including Hoare logic which we use to verify programs, separation logic to model the memory state of programs, and abstract interpretation which we take

as the standard framework to analyse programs. The second part concerns applications of these techniques to verify/analyse programs, including the verification of shape (and related) properties, some program analyses built on the basis of abstract interpretation, shape analysis to infer shape properties about programs, and verification of software systems with unknown components. Finally we will take a note of model checking, which represents another important stream of program verification (yet fairly different from what is used in this thesis).

2.1 Hoare Logic and Program Verification

Hoping to verify that people's programs will run according to their intention, the first two outrunners of Hoare logic are R. W. Floyd and C. A. R. Hoare, with two foundational papers for program verification (Floyd, 1967; Hoare, 1969). They introduce the concept of partial and total correctness and set up the logical base of program verification. Especially the latter one proposes an approach that uses an elucidation of sets of axioms and rules of inference which can be used in deductive reasoning and proofs of the properties of computer programs to set up the axiomatic semantics of computer programs. Therefore such semantics is sometimes called Hoare logic from then on. A later work (Dijkstra, 1976) presents the notion of weakest precondition proven as equivalent as the former. Burstall (1974) integrates operational semantics for programs with the formal verification method mentioned here.

Since then a large number of publications are devoted to Hoare logic. The total correctness version of Hoare calculus is presented in Manna and Pnueli (1974), capable of proving that a program can terminate and is logically correct, which extends Hoare's method by proving correctness and termination at once. The notions of

expressiveness and relative completeness are introduced in Cook (1978), which finds that Hoare logic is only complete in a certain sense, relative to his interpretive semantics. Clarke (1979) researches on the expressiveness of finite interpretations, with the result that certain programming languages can not possess a sound and relatively complete Hoare calculus, because the halting problem is undecidable for the languages, even if the underlying interpretation is finite. Lipton (1977) claims that the only expressive interpretations should be the standard interpretation of Peano arithmetic and the finite interpretation.

The verification and analysis presented in this thesis are essentially founded on the basis of Hoare logic. As will be seen in later chapters, our abstract program semantics used for symbolic executions of programs are based on Hoare logic, or more specifically separation logic (as an extension of Hoare logic), which is surveyed in the next section.

2.2 Separation Logic

For the modelling of program's memory state, we use the technique of separation logic (O'Hearn and Pym, 1999; Reynolds, 1999; O'Hearn et al., 2001; Reynolds, 2002). In this section we will have a brief introduction to its history.

As a prototype of separation logic, O'Hearn and Pym (1999) introduces a logic of bunched implications (BI) which is merged from two parts: additive intuitionistic logic and multiplicative intuitionistic linear logic. Models of propositional BI's proofs are given by bi-cartesian doubly closed categories, combining freely semantics from both logic families. This work also develops a first-order predicate version of BI with newly invented universal and existential quantifiers.

However, BI is no more than a theoretical logic model until Reynolds has presented his work (Reynolds, 1999) to reason about resource-sensitive programs, whose logic model is analogous to BI's. Generally it is an extension of Hoare's approach to proving the correctness of imperative programs that perform destructive updates to data structures containing more than one pointer to the same location. It invents an "independent conjunction" $P \;\&\; Q$ that holds only when P and Q are both true and depend upon distinct areas of storage, whose semantics is exactly the same as the linear conjunction of BI. It is a nice coincidence that they come to the same point from two different ways, which happened several times in the history of computer science such as Turing's computing machine and Church's λ-calculus.

After that these two branches of research group have cooperated to deliver a series of works (O'Hearn et al., 2001; Reynolds, 2002) to set up the foundation of separation logic which can be used to reason about heap memory state. Reynolds (2002) adds two more connectives to classical logic: separation conjunction $*$ and spacial implication $-\!\!*$. The formula $\Delta_1 * \Delta_2$ asserts that two heaps described by Δ_1 and Δ_2 are domain-disjoint, while $\Delta_1 -\!\!* \Delta_2$ asserts that if the current heap is extended with a disjoint heap described by Δ_1, then Δ_2 holds in the extended heap. Such connectives are supported by a low-level storage model based on both the stack and the heap memory. In this model, four sets are assumed: Loc of memory locations, Val of primitive values (with $0 \in \mathsf{Val}$ denoting `null`), Var of variables (program and logical variables), and ObjVal of object values stored in the heap, with $c[f_1 \mapsto \nu_1, \ldots, f_n \mapsto \nu_n]$ denoting an object value of data type c where $\nu_1, \ldots, \nu_n$ are current values of the corresponding fields $f_1, \ldots, f_n$. Then a concrete memory state h, s, consisting of heap and stack, is from the following concrete domains:

$$\begin{aligned} h \in \mathit{Heaps} &=_{df} \mathsf{Loc} \rightharpoonup_{\mathit{fin}} \mathsf{ObjVal} \\ s \in \mathit{Stacks} &=_{df} \mathsf{Var} \to \mathsf{Val} \cup \mathsf{Loc} \end{aligned}$$

This model supports the basic program operations such as lookup, update, allocation and deallocation with a series of Hoare logic style reasoning rules. It also provides

unrestricted memory address arithmetic. In the paper the frame rule

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ \mathsf{mods}(C) \cap \mathsf{fv}(R) = \emptyset$$

is emphasised as the base of *local reasoning*, because it allows the reasoning of programs to concentrate on programs' *footprint*, namely, the variables that the program actually manipulates, instead of a large global heap state. This is important as it entitles the reasoning with the potential to scale up. Separation logic's assertion language is also formalised on a possible worlds model of BI. The soundness and relative completeness are also discussed in the paper, as well as latest results of separation logic with the illustration of its possible applications in the field of program reasoning.

For separation logic itself, there are some other works to address. Yang and O'Hearn (2002) presents a semantic analysis of the soundness and relative completeness of separation logic for the frame axiom to be inferred automatically, with the result that it can be avoided when writing specifications. Calcagno et al. (2001) discusses on some computability and complexity results of separation logic, where it points out that the validity of separation logic formulae is not decidable; however the validity over a restricted subset of separation logic formulae is fortunately decidable with certain complexity. Following it, Berdine et al. (2004) provides a fragment of separation logic whose entailment checking problem is decidable with a sound and complete algorithm to solve it, which plays an important theoretical role in their later works of program analysis. Calcagno et al. (2007) studies the semantic structures lying behind separation logic by the concept of local action, which is a state transformer that mutates the state in a local way. It formulates local actions for a class of models called separation algebras, abstracting from the memory and other specific concrete models used in work on separation logic. Local actions provide a semantics for a generalised form of (sequential) separation logic, and allow a general soundness proof for a separation logic for concurrency.

2.3 Abstract Interpretation

As for the analysis of programs, Cousot has proposed a series of foundational works (Cousot and Cousot, 1976, 1977; Cousot and Halbwachs, 1978) to introduce the framework of abstract interpretation. The first one (Cousot and Cousot, 1976) introduces the basic way to use "abstract" (symbolic) values associated with variables instead of the "concrete" values during real execution of the program. For example, if we choose two domains, one being natural numbers $\mathbb{N}$ as the concrete domain recording concrete values of variables, and the other being the set S of integer intervals as the abstract domain, then we define two functions α and γ as

$$\begin{aligned} \alpha(N) &= [\mathsf{min}(N), \mathsf{max}(N)], N \subseteq \mathbb{N} \\ \gamma(s) &= s, s \subseteq \mathbb{N} \end{aligned}$$

where we write $[n, +\infty)$ as $[n, +\infty]$ for expression convenience. Here we call α the abstraction function and γ the concretisation function, since the first one maps a set of concrete values to an abstract value, and the second one runs in the reversed way. Note that we have the relationship $\forall N \in \mathcal{P}(\mathbb{N}) \cdot N \subseteq \gamma(\alpha(N))$ and $\forall s \in S \cdot s = \alpha(\gamma(s))$. Thus these two functions create a *Galois connection* to link the two domains together. In an analysis when we are confronted with an infinite increasing chain $1, 2, 3, \ldots$ as the value sequence of some variable, we can condense it as an abstract value $[1, +\infty]$ in S to force convergence. The paper then interprets the basic operations of the programming language in this setting accordingly.

On the basis of the first work, the second one (Cousot and Cousot, 1977) proposes an approach to an approximation of fixed-point to construct a unified lattice model for static program analysis. Its general idea is to have some ordering over both concrete and abstract states and an induced (complete) semi-lattice over them, and regard the (recursive) program being analysed as a transition function f, which is monotonic over the concrete domain (and lifted to the abstract domain). Then the

least fixed-point of f ($\mathsf{lfp}\, f$) can be considered as the semantics of f, which may be computed with a fixed-point iteration process.

The third work (Cousot and Halbwachs, 1978) is a utilisation of the first two to discover the assertions (of linear type) that can be deduced from the semantics of the program. It can often discover relations which are never stated explicitly in the program.

After that Cousots still have consequent works to make the framework of abstract interpretation more complete. Cousot and Cousot (1979) exhibits a systematic way to design program analysis frameworks. It shows a (both forward and backward) deductive semantics of programs as the standard of soundness, based on which it studies the design of a space of approximate assertions, and the design of the approximate predicate transformer induced by such assertions. In this way it brings forward some global program analysis methods. This framework is an excellent foundation for other program analysis practice, while its semantics is rectified again in a later work (Cousot, 1981).

For the approximation methods in the fixed-point calculation of abstract interpretation, Cousot and Cousot (1977) also introduces some initial ways which are still frequently referenced today. One is static in that it can be understood as the simplification of the equation involved in the concrete semantics into an approximate abstract equation, whose solution provides the abstract semantics. Galois connections are used in this method to formalise this discrete approximation process. The second is dynamic in that it takes place during the iterative resolution of the abstract equation (or system of equations). This separation introduces additional flexibility allowing for both expressiveness and efficiency. It also introduces the idea of using widening and narrowing operators (∇ and $\triangle$) to accelerate/force convergence for fixed-point approximation (especially when the lattice is of infinite height and does

not satisfy the ascending chain condition). An instance of this follows the previous example of natural real numbers and intervals. For this example we may have a widening operator ∇ by choosing a finite ramp

$$0 = r_0 < r_1 < \cdots < r_k = +\infty$$

and the widening's definition is

$$\begin{aligned} \emptyset \nabla [l', h'] &= [l', h'], \text{ or} \\ [l, h] \nabla [l', h'] &= \begin{array}{l} [\text{if } l' < l \text{ then } \mathsf{max}\{r_i | r_i \leq l'\} \text{ else } l, \\ \quad \text{if } h' > h \text{ then } \mathsf{min}\{r_i | h' \leq r_i\} \text{ else } h\,] \end{array} \end{aligned}$$

such that if we choose $r_{k-1} = 1$, then when we have an infinite ascending chain during the analysis $[0,0], [0,1], [0,2], \ldots$ we can use ∇ to widen each state with its consecutive state in order to make the chain converge as $[0,0], [0,1], [0,+\infty]$. This idea is essential in the work because it offers a way to deal with infinite lattices not satisfying the ascending chain condition or to speed up convergence in case of combinatorial explosion.

In this thesis we also apply a technique related to abstract interpretation to implement our analysis, viz. *constraint abstraction* (Gustavsson and Svenningsson, 2001). Its kernel idea is to view the program being analysed as a transition function (constraint), then abstract this function as an appropriate form according to the program's denotational semantics, and solve the obtained constraint abstraction on the program state lattice to get the analysis result. For example, for a while loop

$$\texttt{while(x} > \texttt{0)}\ \{\, \texttt{x} = \texttt{x} - \texttt{1};\ \texttt{y} = \texttt{y} + \texttt{1};\, \}$$

to start in a state $\{\texttt{x}{>}\texttt{0} \wedge \texttt{y}{=}\texttt{0}\}$, we can utilise constraint abstraction to infer its invariant, as the invariant reflects the transition from the initial state to the current state by the while loop. We denote the constraint abstraction representing this loop as $\texttt{Q}(\texttt{x}, \texttt{y}, \texttt{x}', \texttt{y}')$ where $\texttt{x}$ and $\texttt{y}$ are initial values of variables $\texttt{x}$ and $\texttt{y}$, and $\texttt{x}'$ and $\texttt{y}'$ are their current values, then $\texttt{Q}$ can be computed from the loop body as

$$\texttt{Q}(\texttt{x}, \texttt{y}, \texttt{x}', \texttt{y}') \equiv (\texttt{x}{\leq}\texttt{0} \wedge \texttt{x}{=}\texttt{x}' \wedge \texttt{y}{=}\texttt{y}') \vee (\texttt{x}{>}\texttt{0} \wedge \texttt{Q}(\texttt{x}{-}\texttt{1}, \texttt{y}{+}\texttt{1}, \texttt{x}', \texttt{y}'))$$

where we can observe that $\mathtt{Q}$ is inductively defined on itself, suggesting the recursive nature of the while loop. For this constraint abstraction, we may execute a standard fixed-point iteration process armed with widening operations from Cousot and Cousot (1977) to find one of its fixed-points as $\mathtt{x}-\mathtt{x}' = \mathtt{y}'-\mathtt{y}$, which is exactly the loop invariant that we are after.

2.4 Verification of Shape Properties

From this section on we will turn away from techniques we use to relevant works of this thesis. Our aim of research is to provide a verification system to verify shape and relevant properties of heap-manipulating programs with partial information. In this field there have been active similar efforts, especially for the verification and analysis of memory consumption and manipulation, as discussed below.

Smallfoot (Berdine et al., 2005b) is a verification tool based on separation logic. It makes use of a symbolic execution designed to work with a fixed set of shape predicates, most notably the list segment predicate (Berdine et al., 2005a). It is the first attempt to use separation logic in the verification of pointer safety and simple shape properties. The data structures and properties that they verify are quite simple compared with our user-defined predicates.

Another work on this aspect, Java Modelling Language (JML), is a behavioural interface specification language designed to specify the behaviours and interfaces of Java programs (Leavens et al., 2000; Burdy et al., 2005; Leavens et al., 2006). The design of JML is heavily influenced by the model-based approach of Larch, in particular the Larch/C++ works by Leavens and others (Cheon and Leavens, 1994; Leavens and Baker, 1999). The language has attracted significant research

attention, resulting in a number of tools supporting JML, ranging from run-time assertion checking tools (Cheon, 2003) to static time verifiers (Cataño and Huisman, 2003; Cok and Kiniry, 2004). There have also been efforts to apply JML on a larger scale (Poll et al., 2001; Schmitt et al., 2006).

For the features of JML, its specifications, which consist of Java expressions and some additional keywords, are embedded as specially-formatted comments in Java source code. JML supports specification constructs such as model fields, ghost fields, and model methods. A model field is an abstract representation of some concrete states; the link between these two are given using the JML's `represents` clause, which syntactically distinguishes whether the relationship is functional, or relational. Our approach, on the other hand, uses predicate parameters in lieu of specification variables. The dependency of the predicate parameters and concrete program states are captured by the predicate definitions, which can capture both functional and relational dependencies in a uniform manner.

Extended Static Checking for Java (ESC/Java) (Flanagan et al., 2002), developed at Compaq Systems Research Centre, aims to detect more errors than "traditional" static checking tools, but is not designed to be a program verification system. The stated goals of ESC/Java are scalability and usability. For that, it forgoes soundness for the potential benefits of more automation and faster verification time. However, it is not a sound checker in the sense that it may miss errors, i.e. it cannot guarantee the absence of certain class of errors. The ESC/Java effort is continued with ESC/Java2 (Cok and Kiniry, 2004), which adds support for current versions of Java, and verifies more JML constructs. One significant addition is the support for model fields and method calls within annotations (Cok, 2005).

Spec$^\sharp$ (Barnett et al., 2004b) is a programming system developed at Microsoft Research. It is an attempt at verifying programs written in the C$^\sharp$ programming

language. It adds constructs tailored to program verification such as pre- and post-conditions, frame conditions, non-null types, object invariants, etc. Spec$^\sharp$ programs are verified by the Boogie verifier (Barnett et al., 2006), which uses Simplify (Detlefs et al., 2005) and Z3 (de Moura and Bjørner, 2008) to discharge its proof obligations. Spec$^\sharp$ also supports runtime assertion checking.

Spec$^\sharp$ supports object invariants but leaves the decision of when to enforce/assume object invariants to the user. It adds special fields to each object, which can be mentioned explicitly in method pre- and post-conditions. The values of these fields determine if the object invariant is enforced by the corresponding method contract (Barnett et al., 2004a). In order to verify object invariant modularly, Spec$^\sharp$ employs an ownership scheme that allows an object o to own its representation —- objects that are reachable from o and are part of o's abstract state. The ownership scheme in Spec$^\sharp$ forces a top-down unpacking of the objects for updates, and a bottom-up packing for re-establishing the object invariant. The packing and unpacking of objects are done explicitly by having programmers writing special commands in method bodies.

There is yet another success in shape analysis and verification, that is, Hob and Jahob. The former one is designed to verify data structure consistency properties (Lam, 2007), which incorporates multiple analyses, called analysis plug-ins, to verify diverse properties of global data structures (Kuncak et al., 2006). Different analysis plug-ins communicate program states by using boolean algebra of sets with cardinality constraints. The language is expressive enough to allow Hob to encode and verify a number of application-specific properties. Apart from this common specification language, each analysis plug-in maintains its own internal representation of program states. Hob provides a way for the plug-ins to link their own representation with the common abstraction by using abstraction modules. The analysis-plug-in architecture has the important flexibility of allowing different

analyses with different precision/scalability trade-offs to work on different modules of the same program, which could open up the possibility of having larger programs verified.

Jahob (Kuncak et al., 2006; Kuncak, 2007) continues Hob's effort in using different analyses to verify complex properties of linked data structures. However, Jahob deviates from Hob in some important aspects. Instead of using specially-designed implementation and specification languages like Hob does, it uses a subset of Java as its implementation language and a subset of the Isabelle/HOL (Nipkow et al., 2002) language as its specification language; it also works on instantiable data structures, as opposed to Hob's global data structures. Jahob develops a technique to combine multiple theorem provers (Klarlund and Møller, 2001; Nipkow et al., 2002; Detlefs et al., 2005) to reason about expressive logical formulas in Higher Order Logic (HOL). It makes full use of each theorem prover's advantageous point, which is the key feature of this verification system.

Finally, our work is based on the improvement of another state-of-the-art program verifier Hip/Sleek (Nguyen et al., 2007; Chin et al., 2007; Nguyen and Chin, 2008). Its overview is given in Figure 2.1. The front-end of the system is a standard Hoare-style forward verifier Hip, which invokes the separation logic prover Sleek. The Hip verifier comprises a set of forward verification rules to systematically check that the precondition is satisfied at each call site, and that the declared postcondition is successfully verified (assuming the given precondition) for each method definition. For the separation logic prover Sleek, given two program states Δ_1 and Δ_2, it attempts to prove that Δ_1 entails Δ_2; if it succeeds, it returns a frame $\mathtt{R}$ such that $\Delta_1 \vdash \Delta_2 * \mathtt{R}$. As discussed previously, it can express and process multiple types of program properties such as shape, quantitative and content ones in program states Δ. We want to keep all its merits and improve it by dealing with partial information in verification. Meanwhile, we also use the Sleek tool as our main

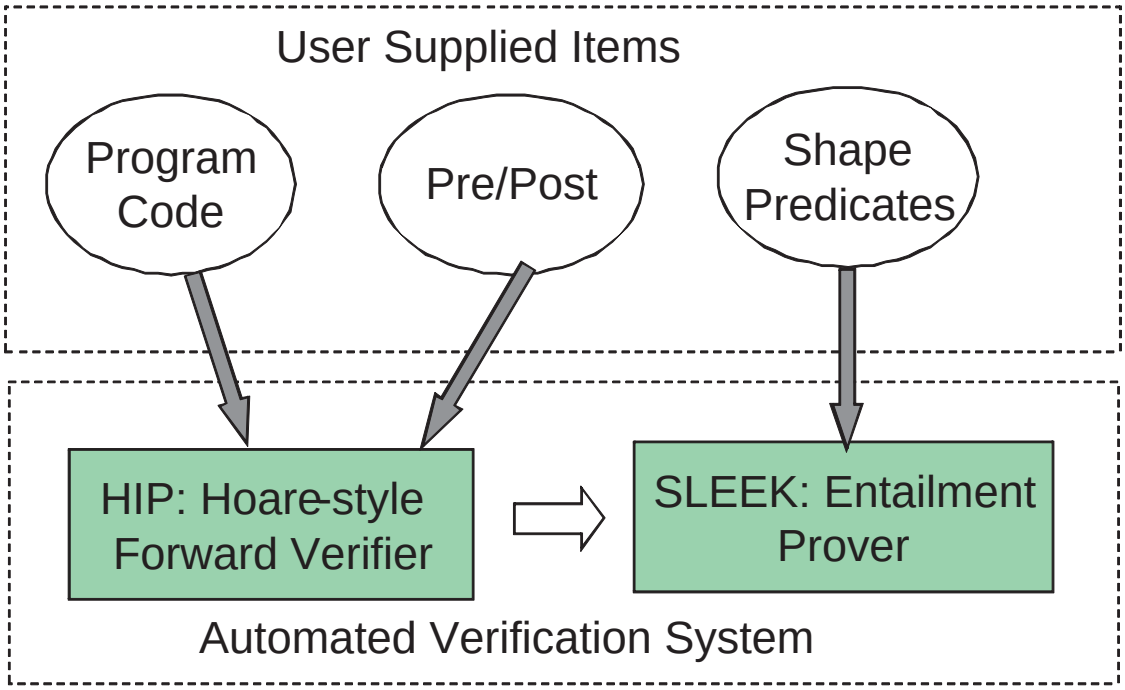

Figure 2.1: The HIP/SLEEK verification system.

solver for entailment checking.

Our project will be different from the aforementioned ones in the techniques employed (like separation logic and entailment checking) in that we can describe the shape properties in a more natural way from the user's perspective and still remain expressive and computationally feasible. Meanwhile we also try to verify programs that are provided with only partial information by inferring constraints to complete the partial specification and to describe the missing part of the partial program.

2.5 Program Analysis for General Program Properties

To automate the verification process, program analysis techniques are widely applied on the basis of abstract interpretation. This section surveys those analyses for general program properties, such as numerical properties like the value range of program variables. The pioneer work in this field is what we have already introduced (Cousot and Cousot, 1976). It begins with the basic "program units" to study each's behaviour in an intra-procedural analysis, and then the behaviour among different procedures is put into consideration to form the inter-procedural analysis. In this way it initialises the research in this area. Next we will survey some more recent ones.

Reps et al. (1995) shows how to analyse programs precisely with finite abstract domains and distributive transfer functions. Its means is to transform such dataflow analysis problems into a special kind of graph reachability problem. It promises precision while also controls the time complexity in polynomial scale.

For more recent works, Müller-Olm and Seidl (2004); Popeea and Chin (2006); Gulwani and Tiwari (2007) present precise inter-procedural analyses with linear equalities. The first (Müller-Olm and Seidl, 2004) applies linear algebra techniques to precise dataflow analysis to describe analyses determined for each program point identities that are valid among the program variables whenever control reaches that program point (as their main approach). They fully interpret assignment statements with affine expressions on the right hand side to compute the set of all affine relations and polynomial relations of bounded degree. Their complexity is worth noting to be linear to program size and polynomial to the number of variables. The second (Popeea and Chin, 2006) also introduces the notion of affinity to characterise

how closely related a pair of polyhedra is. Then they try to find related elements in the polyhedron (base) domain to allow the formulation of precise hull and widening operators lifted to the disjunctive (powerset extension of the) polyhedron domain. In this way they effectively prevent the original convex-hull's loss of precision. The third (Gulwani and Tiwari, 2007) uses a backward analysis to propagate information with "generic assertions" facilitating inter-procedural analysis and simplify such assertions with unification. Their analysis and implementation are constructed on this technique. In our work, we can utilise their achievements as solvers to some part of our combined domain, for example, the numerical constraint abstraction generated from the original program we want to verify.

2.6 Pointer, Alias and Shape Analysis

As a significant branch of program analysis, the analysis of pointer and memory safety became a heated research topic in early 1990s (Landi and Ryder, 1992; Choi et al., 1993). Pointer analysis is a static code analysis technique that establishes which pointers, or heap references, can point to which variables or storage locations, and such result may be used in a program verification for pointer-related properties. Pointer analysis itself still has several branches such as alias analysis (Emami, 1993) and shape analysis (Sagiv et al., 2002), which are investigated in sequence.

For aliasing analyses, Chatterjee et al. (1999); Cheng and Hwu (2000) propose two modular approaches. The first (Chatterjee et al., 1999) presents a modular technique for flow-sensitive and context-sensitive dataflow analysis of statically typed object-oriented programming languages such as C++ and Java, namely, relevant context inference. It can be used to analyse complete programs as well as partial programs such as libraries and is totally statical. The main part of their method is

a modular way of analysing the method bodies separately before the calling context information, while they also study all the valid program paths in an analysis. The second work (Cheng and Hwu, 2000) is similar in the light that it uses access paths as well to perform a modular pointer analysis, while it addresses the context in the analysis, including program flow and context sensitivity, to avoid the overhead of representing context-sensitive transfer functions.

Recently Yorsh et al. (2008) shows how to combine finite type state with aliasing analysis. It advocates that their framework for generating procedure summaries is both precise (applying the summary in a given context yields the same result as re-analysing the procedure in that context) and concise (the summary exploits the commonalities in the exact ways the procedure manipulates abstract values). For this sake they define a class of abstract domains and transformers, which can be instantiated to perform the analysis.

In the field of shape analysis, there are also a series of distinguished works. Sagiv et al. (1999, 2002) represent the initial works on this aspect. It is a summating work as it provides a parametric framework for shape analysis that can be instantiated in different ways to create different shape analysis algorithms that provide varying degrees of efficiency and precision. Many works have been built on this framework, such as Möller and Schwartzbach (2001) and Wies et al. (2007), etc. It is also extended by Lee et al. (2005) which exhibits a grammar-based shape analysis with grammar annotations in order to express the shape of complex data structures precisely.

Rinetzky et al. (2005) presents a shape analysis method that uses a characterisation of a procedure's behaviour in which parts of the heap not relevant to the procedure are ignored. Actually this shares the same idea as separation logic does, but its semantics is $\mathcal{LSL}$, and their static analysis algorithm is founded on the abstract

interpretation of $\mathcal{LSL}$.

Similar as our motivation, Lahiri and Qadeer (2006) bases shape analysis on verification. Its main technical contribution is a novel method for verifying linked lists based on two new predicates that characterise reachability of heap cells; these two predicates also allow reasoning about both acyclic and cyclic lists. Meanwhile, it proposes a concept of well-foundedness which is useful for our preceding work Hip/Sleek (Nguyen et al., 2007).

Hackett and Rugina (2005) uses local reasoning about individual heap locations, instead of global reasoning about entire heap abstractions. They build the shape abstraction and analysis on top of a pointer analysis, and decompose the shape abstraction into a set of independent configurations, each of which characterises one single heap location. One key feature of their analysis is that it can be used to enable the static detection of memory errors in programs with explicit deallocation.

Manevich (2009) presents some new "partially disjunctive" shape analyses aimed at taming the size of the state space by abstracting disjunctions, as well as soundly approximating program statements. It implements and applies these analyses to prove properties of sequential programs and fine-grained concurrent programs, with respect to several properties, including cleanness properties, shape invariants, and linearisability of concurrent data structure implementations.

Another important stream of analysis work done in separation logic is Berdine et al. (2005a); Distefano et al. (2006); Gotsman et al. (2006); Yang et al. (2008); Calcagno et al. (2009). Among them, Distefano et al. (2006) follows the method of Smallfoot's symbolic execution of certain separation logic formulae called symbolic heaps (Berdine et al., 2005a). As a result, it also uses the linked list and its segment predicates, as in their former verification work, to perform intra-procedure inference

on while-loops. Gotsman et al. (2006) proposes an inter-procedural shape analysis that makes use of spatial locality (i.e. the fact that most procedures modify only a small subset of the heap) in its representation of abstract states. In fact they uses separation logic itself to express the shape properties. However, they only use the original pointing-to assertion provided by separation logic instead of one more level of abstraction. They track reachability information indirectly and aliasing information directly, which is an important feature of their work. Following these works, two other significant papers (Yang et al., 2008; Calcagno et al., 2009) prove the feasibility of their approaches by successfully analysing parts of medium- and large-size scales software, including device drivers and system software such as OpenSSH and Linux distributions. They accomplished these objectives by tuning the join operator used in the analysis and applying abductive reasoning to discover full specifications for programs.

There are also works addressing program analyses for both shape and other program properties on the basis of separation logic. THOR (Magill et al., 2007, 2008) advocates to separate the analysis of shape properties from that of other properties, by deriving an approximated numerical program from the original heap-manipulating program. This idea is similar as this thesis. In the development of the THOR tool, Magill et al. (2007, 2008) propose an adaptive shape analysis where additional numerical analysis can be used to help gain better precision. Its abstraction mechanism is also employed in C-to-gate hardware synthesis (Cook et al., 2009). Very recently, Magill et al. (2010) formulates a novel instrumentation process which inserts numerical instructions into programs, based on their shape analysis and user-provided predicates. Instrumented programs can then be used to generate pure numerical programs for further analysis.

With respect to the content of data structure, Ireland (2007) proposes a cooperative approach. In that work the author applies the symbolic evaluation technique

of THOR's, and specifies the loop invariant as a combination of two parts: the shape part and the schematic content part. This approach concentrates more on the automation of the analysis, at the cost of generating possibly unsound invariants. Therefore a middle-out proof planning procedure is employed to rule out such invariants. As an improvement, Maclean et al. (2009) attempts to refine the obtained invariants and uses IsaPlanner (Dixon, 2005) to handle meta-variables and goal-naming issues. With these techniques it can perform analysis (for instance over its illustrative array example) for both memory safety and functional correctness.

Combining the two features above together, Bouajjani et al. (2010) can handle both numerical and (restricted) content of linked lists at the same time. They synthesise list-related invariants over infinite data domains using graph heap representation. The synthesised invariants are able to capture various aspects of data structures, such as the size, the sum or the multiset of linked list, relations of the data at linearly ordered or successive positions.

The above groups of works mainly focus on list data structure. For non-linear data structures, Chang et al. (2007); Chang and Rival (2008) design an abstract domain in separation logic that is parameterised both by an abstract domain for pure data properties and by user-supplied specifications of the data structure invariants to check. It supports various types of invariants about shape and data and features a mechanism for materialising summaries. Upon this domain, they build a shape analysis using abstract interpretation and a widening operator over the combined shape and data domain.

Compared in general with the works stated above, this thesis focuses more on the verification of shape and its relevant properties, whereas it also exploits shape analysis techniques to deal with the partial information in target programs. To handle memory safety as well as functional correctness of heap-manipulating programs, we

work on a combined abstract domain which is more complex than previous works, and we aim to infer both preconditions and postconditions for methods starting from partial information, which is outstanding from the aforementioned results.

2.7 Verification of Programs with Unknown Components

Program analysis techniques are employed to discover unknown program specifications/invariants, while there are also techniques to verify programs with unknown components, where some program code is not available to the verifier.

Black-box testing (Beizer, 1996) views the unknown components in programs as black-boxes to test their functionality. It has certain patterns for users to design pairs of input and output, and use them to test whether the program meets people's expectations. This approach is now widely applied in software industry; however, in essence it is a method of testing instead of verification; therefore it cannot formally prove the absence of program bugs. Especially in safety-critical systems a bug failed to be found by such testing may cause catastrophic result, as described in Chapter 1. The same problem also applies to some similar approaches like specification mining (Ammons et al., 2002). For unknown components of a program, It discovers possible specifications for them by observing the program's execution and traces, which is also dynamically performed and bears the same non-exhaustion problem.

For such problem, static verifiers/analysers are more proficient, as they can explore all possible program states. However, for programs with unknown components, existing verifiers/analysers usually do not perform very well. For example, SpaceInvader (Calcagno et al., 2009) simply assumes the program and the unknown

procedure have disjoint memory footprints so that the unknown call can be safely ignored due to the hypothetical frame rule (O'Hearn et al., 2004), whereas this assumption does not hold in many cases. Some other verification approaches (Emami et al., 1994; Gopan and Reps, 2007) attempt to take into account all possible implementations for the unknown component. The first one (Emami et al., 1994) is founded on a points-to analysis with context-sensitive inter-procedural information which captures all calling contexts. The second one (Gopan and Reps, 2007) looks at library functions' low-level implementation to construct summary information for linked libraries with no source code available. However, for these methods, there can be too many such candidates in general, and hence the verification might be infeasible for large-scaled programs.

It is also notable that the similar problem is addressed by some model checking works. (Model checking is yet another stream of program correctness proof which will be introduced in the next section.) Peled et al. (1999) tries to test whether an implementation with unknown structure satisfies some given properties, and proposes an approach by learning and adjusting the checking process via experiments, known as black box checking. Li et al. (2002) represents a decompositional approach able to check feature-oriented (open) software designs which may cause false alarms. Xie and Dang (2005) is also a decompositional method integrating features from both testing and model checking. They reduce the testing of the global system down to the testing of each black-box, and generate test sequences for each individual box using an automata-theoretic approach. Their method is sound and complete, but the properties that it may test are very limited.

Compared with the approaches above, our work is exhaustive in finding a specific type of program bugs, is sound with respect to program semantics, and meanwhile still have the potential to scale up because we do not consider all possible implementations to an unknown component. The program properties that we focus on include

strong program invariants about data structure's shape, size, relational information and content, which are rather difficult for existing tools that can verify programs with unknown components.

2.8 Model Checking

Model checking (Clarke and Emerson, 1981) represents a fairly different approach to the proof of software correctness. It was originally designed to verify finite-state systems by exhausting the whole set of computation states according to some specification described in temporal logic, and achieved great success on circuit design and implementation (McMillan, 1992). Such success intrigued researchers' interest in applying model checking to the field of software. The key technique for such application is abstraction (like predicate abstraction (Ball et al., 2001) and counterexample-guided abstraction refinement (Clarke et al., 2000, 2003)), as software usually has infinite computation states which are beyond the capability of model checking. These abstractions are even borrowed in some analysis works, such as Sagiv et al. (2002); Balaban et al. (2005). With appropriate abstraction techniques, model checking tools are generally automatic and thus requires no user intervention. Some representatives of such tools include the general framework of model checkers SPIN (Holzmann, 2004), the SLAM model checker for drivers (Ball and Rajamani, 2001), the BLAST model checker for C programs with lazy counterexample-guided abstraction refinement (Henzinger et al., 2003) and Java PathFinder for Java programs by NASA (Visser et al., 2003). In this thesis, we employ a slightly different Hoare logic based approach to verify the program properties (including shape and others) with user-provided predicates of abstraction and program analysis techniques.

2.9 Summary

This chapter has surveyed the state-of-the-art in the field of software quality assurance, especially program verification and analysis. It mainly covers two types of topics, one being the foundational techniques that we use in this thesis, and the other being some related works by peers in the similar area of program verification and analysis.

Chapter 3

Language and Semantics

In this chapter our programming and specification languages are introduced together with their semantics. We use a standard object-based programming language (to write the programs being verified) and a predicate-based specification language (to express program specifications and abstract program states). Our predicates are inductively defined as in HIP/SLEEK (Nguyen et al., 2007; Chin et al., 2007; Nguyen and Chin, 2008; Chin et al., 2010), and can capture recursive data structures with sophisticated program properties involving not only structural aspects but also quantitative aspects as well as content of data structures. In what follows, we will first define the programming language, then give a brief introduction to the separation logic we use as a base for our specification language, and finally design the specification language itself.

3.1 Programming Language

3.1.1 Grammar

The programming language used in our system is a typed object-based language which may be viewed as a subset of popular type-safe programming languages such as Java or C♯. Its grammar is formally defined in Figure 3.1.

A program *Prog* in our language consists of a list of type declarations *tdecl* and a list of procedure definitions *meth*. The type declarations include class types *classt* used in programs, user-defined predicates *spred* for specifications, and lemmas *lemma*. (We will leave *spred* and *lemma* until Section 3.3 as they are more for the specification language.) Compared with those fully-fledged object-oriented languages, our language has omitted some features orthogonal to this thesis' interest, such as inheritance and dynamic dispatch, concurrency, array, exception, and so on. The semantics of most constructs of the language are understood in the usual sense that one would find in languages such as Java or C♯, except for the class declaration, which declares a class type without instance procedures or dynamic dispatch. Other than that, they behave like normal classes: instances of a class type can be (dynamically) allocated, their fields read (`v.f`) and updated (`v.f = w`), references to them passed to and from procedures, etc. (In this sense they are more like a `struct` in the C language.) A type t can either be a class type or a primitive built-in type.

For the users to express program specifications, our language includes annotations for procedures and loops. Each procedure *meth* is decorated with its specifications *mspec*. Note that annotations for loops are not written in the traditional style of loop invariants, but rather in the same way as procedure contracts (`where` $\Phi_{pr} \ast\!\!\rightarrow \Phi_{po}$). This facilitates our conversion from loops to tail-recursive procedures so as to treat

Program	*Prog*	::=	$\boldsymbol{tdecl\ meth}$	
Type declaration	*tdecl*	::=	*classt* | *spred* | *lemma*	
Class declaration	*classt*	::=	`class` c { $\boldsymbol{field}$ }	
Field declaration	*field*	::=	$t\ v$	
Type	t	::=	$c \mid \tau$	
Procedure declaration	*meth*	::=	$t\ mn\ ((\boldsymbol{t\ v}); (\boldsymbol{t\ v}))\ \boldsymbol{mspec}\ \{e\}$	
Built-in type	τ	::=	`int` | `bool` | `void`	
Expression	e	::=	d	heap-insensitive atomic
		|	$d[v]$	heap-sensitive atomic
		|	$v{=}e$	assignment
		|	$e_1; e_2$	sequence
		|	$t\ v;\ e$	local variable
		|	`if` $(v)\ e_1$ `else` e_2	
		|	`while` $v\ \{e\}$ `where` $\Phi_{pr} \ast\!\!\rightarrow \Phi_{po}$	
Heap-insensitive atomic	d	::=	-	skip
		|	`null`	null reference
		|	k^{τ}	constant
		|	v	variable
		|	`new` $c(\boldsymbol{v})$	allocation
		|	$mn(\boldsymbol{u}; \boldsymbol{v})$	method call
Heap-sensitive atomic	$d[v]$	::=	$v.f$	field read
		|	$v.f{=}w$	field write
		|	`free`(v)	deallocation

Figure 3.1: A core (Java-like) imperative language.

them in the same manner as normal procedures. For this sake, we support both pass-by-value and pass-by-reference parameters, which are separated with a semicolon ; where the ones before ; are pass-by-value and the ones after are pass-by-reference. The grammar for these annotations will be presented in Section 3.3. The meaning of a pair of precondition and postcondition is that if the procedure is invoked in a program state satisfying its precondition, the procedure will not have any memory faults such as null or dangling pointer dereferences. Furthermore, if the procedure terminates, it terminates in a state that satisfies the postcondition. Otherwise, if the program state does not satisfy the precondition, then the verification fails and a catastrophic error is reported with its location in the program. In other words, we adopt the partial correctness semantics of Hoare triples with tight interpretation (Yang and O'Hearn, 2002).

Without loss of generality, our language is expression-oriented, so the body of a method is an expression composed of standard instructions and constructors of the language. e is the (recursively defined) program constructor, and d and $d[v]$ are atomic instructions. Here $d[v]$ has some specific requirement over the memory state (such that v must be allocated at a valid part of heap memory) and is therefore named heap-sensitive atomic instruction, whereas d does not have such requirements and is called heap-insensitive atomic instruction. As will be seen in later chapters, these two sorts of instructions are treated differently during the analysis of a program.

We have some further assumptions over the programs, so that they are well-formed according to the following rules. Each program should be type-safe. Classes, predicates and procedures should have distinct names. Local variables in the same scope are distinct. Meanwhile, we do not allow the syntactic sugar for local variables to hide variables from outer scopes or procedure parameters.

3.1.2 Operational Semantics

This section defines the operational semantics of our programming language. Before doing that, we first define the semantic domains. Locations in our system correspond to object identifiers (which can be practically regarded as memory locations). Values include primitive values, locations, and the special value `null` that does not correspond to any object identifier. Objects are finite partial maps that map field names to values. Primitive values include integer numbers and boolean values.

The operational semantics for our language is a small-step semantics which are essentially transitions between machine configurations. Each machine configuration is a triple consisting of:

- Heap h. We model heaps as finite partial maps from locations to objects. Objects are expected to conform to their defined class types.
- Stack s. Stacks are modelled as finite partial maps from variables to values. Note that it is viewed as a "stackable" mapping, where a variable v may occur several times, and $s(v)$ always refers to the value of the variable v that was popped in most recently.[1]
- Current program code e. Program execution terminates when e is -, a value of type `void`.

For simplicity, we assume that all `while` loops are already transformed to tail-recursive methods with pass-by-reference parameters. Each reduction step can then

[1] A more formal definition for s would mark different occurrences of the same variable with different "frame" numbers. We omit the details here.

be formalised as a small-step transition of the form:

$$\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$$

The full set of transitions is given in Figure 3.2. We explain some of the notations used in them. The operation $[v \mapsto \nu] + s$ "pushes" the variable v to s with the value ν, and $([v \mapsto \nu] + s)(v) = \nu$. The operation $s - \boldsymbol{v}$ "pops out" variables $\boldsymbol{v}$ from the stack s. $s[v \mapsto k]$ is a mapping which keeps all the mappings in s except that of v (which is now specified to be mapped to k). We also abuse this notation for a class type identifier c to denote a region of heap (mappings) in the form $c[f_1 \mapsto s(v_1), \ldots, f_n \mapsto s(v_n)]$, which is essentially a heap location where fields f_i are further mapped to values $s(v_i)$, $i = 1, \ldots, n$. $\perp$ represents an arbitrary value. We also introduce an intermediate construct as results returned by expressions/method calls $\texttt{ret}(\boldsymbol{v}, e)$, where $\boldsymbol{v}$ will be dropped from s after the evaluation/invocation of e, to simulate the behaviour of stack. Whenever such a result is yielded, we assume it is stored in a special logical variable `res`, although `res` is never explicitly put in the stack s.

3.2 Separation Logic

Our specification language is built on top of separation logic (O'Hearn and Pym, 1999; Reynolds, 1999; Ishtiaq and O'Hearn, 2001; Reynolds, 2002), designed for reasoning about programs that manipulate shared mutable pointer-based data structures. The distinguished feature of separation logic is its *local reasoning* about data structures linked with pointers and allocated in heap (Distefano et al., 2006). It means that reasoning about a command concerns only the part of the heap that the command accesses, a.k.a. the command's footprint. Note that local reasoning is not registered patent for separation logic; it also exists in the original formulation of Hoare logic (Hoare, 1969) with the substitution treatment in assignment. However,

OS-VAR $\langle s, h, v\rangle \hookrightarrow \langle s, h, s(v)\rangle$

OS-CONST $\langle s, h, k\rangle \hookrightarrow \langle s, h, k\rangle$

OS-SEQ $\langle s, h, \text{-}; e\rangle \hookrightarrow \langle s, h, e\rangle$

OS-ASSIGN-1 $\langle s, h, v{=}k\rangle \hookrightarrow \langle s[v{\mapsto}k], h, \text{-}\rangle$

OS-FIELD-READ $\langle s, h, v.f\rangle \hookrightarrow \langle s, h, h(s(v))(f)\rangle$

OS-LOCAL $\langle s, h, \{t\ v;\ e\}\rangle \hookrightarrow \langle [v{\mapsto}\bot]{+}s, h, \texttt{ret}(v, e)\rangle$

OS-RET-1 $\langle s, h, \texttt{ret}(\boldsymbol{v}, k)\rangle \hookrightarrow \langle s{-}\{\boldsymbol{v}\}, h, k\rangle$

OS-PROG $$\frac{\langle s, h, e_1\rangle \hookrightarrow \langle s_1, h_1, e_3\rangle}{\langle s, h, e_1; e_2\rangle \hookrightarrow \langle s_1, h_1, e_3; e_2\rangle}$$

OS-ASSIGN-2 $$\frac{\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle}{\langle s, h, v{=}e\rangle \hookrightarrow \langle s_1, h_1, v{=}e_1\rangle}$$

OS-RET-2 $$\frac{\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle}{\langle s, h, \texttt{ret}(\boldsymbol{v}, e)\rangle \hookrightarrow \langle s_1, h_1, \texttt{ret}(\boldsymbol{v}, e_1)\rangle}$$

OS-FIELD-WRITE $$\frac{r = h(s(v_1))[f{\mapsto}s(v_2)] \quad h_1 = h[s(v_1){\mapsto}r]}{\langle s, h, v_1.f = v_2\rangle \hookrightarrow \langle (s, h_1, \text{-})\rangle}$$

OS-IF-1 $$\frac{s(v){=}\texttt{true}}{\langle s, h, \texttt{if}\ (v)\ e_1\ \texttt{else}\ e_2\rangle \hookrightarrow \langle s, h, e_1\rangle}$$

OS-IF-2 $$\frac{s(v){=}\texttt{false}}{\langle s, h, \texttt{if}\ (v)\ e_1\ \texttt{else}\ e_2\rangle \hookrightarrow \langle s, h, e_2\rangle}$$

OS-NEW $$\frac{\begin{array}{c}\texttt{class}\ c\ \{t_1\ f_1, .., t_n\ f_n\} \\ \iota{\notin}dom(h) \quad r{=}c[f_1{\mapsto}s(v_1), .., f_n{\mapsto}s(v_n)]\end{array}}{\langle s, h, \texttt{new}\ c(\boldsymbol{v})\rangle \hookrightarrow \langle s, h[\iota \mapsto r], \iota\rangle}$$

OS-CALL $$\frac{\begin{array}{c}s_1{=}[w_i{\mapsto}s(v_i)]_{i=1}^{m-1} + s \\ t_0\ mn((t_i\ w_i)_{i=1}^{m-1}; (t_i\ w_i)_{i=m}^{n})\ \{e\}\end{array}}{\langle s, h, mn(\boldsymbol{v})\rangle \hookrightarrow \langle s_1, h, \texttt{ret}(\{w_i\}_{i=1}^{m-1}, [v_i/w_i]_{i=m}^{n}e)\rangle}$$

Figure 3.2: Operational semantics.

such local reasoning is lost if heap-based data structure and aliasing are introduced to the programming language. This loss of locality is noted as the pointer swing problem by Hoare and He (1999). In this scenario, separation logic restores the capability to reason locally by means of two technical novelties: 1) the separation conjunction $*$ and 2) tight interpretation of Hoare triples (Yang and O'Hearn, 2002). A key insight leading to separation logic is that program logics for reasoning about heap-manipulating programs should be explicit about the heap. In other words, program heaps should be part of the model of a program logic. The satisfiability of a separation logic formula Δ in a program state is thus typically enforced by the semantics relation

$$s, h \models \Delta$$

where s is a model of the program stack, h the program heap.

Some novel notations that separation logic has introduced include the points-to relationship $\mapsto$ and empty heap `emp`, and two new connectives (separation conjunction $*$ and spatial implication $-\!\!*$). A points-to formula $\mathtt{x} \mapsto \mathtt{y}$ describes a singleton heap with only one cell at address `x` that stores value `y`. Formula `emp` holds on empty heaps. Formula $\Delta_1 * \Delta_2$ describes a heap that can be partitioned into two domain-disjoint heaps described by Δ_1 and Δ_2. Formula $\Delta_1 -\!\!* \Delta_2$ describes a heap that if extended with a disjoint heap represented by Δ_1, then Δ_2 holds in the extended heap. In other words, $\Delta_1 -\!\!* \Delta_2$ captures the heap described by Δ_2, where the heap corresponding to Δ_1 is "taken away". The formal semantics of these operators will be defined formally in Section 3.3.4.

Tight interpretation is another key aspect of separation logic, which ensures that "well-specified programs do not go wrong" (Reynolds, 2005). Under this interpretation, a valid Hoare triple $\{\Delta_1\}\ e\ \{\Delta_2\}$ guarantees that command e should never encounter a memory fault if started in a program state satisfying Δ_1. One significant prerequisite of this interpretation requires the precondition Δ_1 of a command

to guarantee that all memory locations accessed by the command, except for the freshly allocated ones, are allocated beforehand. In the setting of separation logic, a memory location $\mathtt{x}$ is considered allocated if the points-to fact $\mathtt{x} \mapsto \_$ is present. More specifically, Hoare triples for heap-accessing commands in separation logic are as follows:

- Field read:

$$\{\mathtt{x} \mapsto [\mathtt{v}_1, \ldots, \mathtt{v}_i, \ldots, \mathtt{v}_n]\}\ \mathtt{x.f}_i\ \{\mathtt{x} \mapsto [\mathtt{v}_1, \ldots, \mathtt{v}_i, \ldots, \mathtt{v}_n] \wedge \mathtt{res}{=}\mathtt{v}_i\}$$

  where $\mathtt{res}$ is the special variable denoting the resulted value of an expression.

- Field write:

$$\{\mathtt{x} \mapsto [\mathtt{v}_1, ..., \mathtt{v}_{i-1}, \mathtt{v}_i, \mathtt{v}_{i+1}, ..., \mathtt{v}_n]\}\ \mathtt{x.f}_i{=}\mathtt{v}\ \{\mathtt{x} \mapsto [\mathtt{v}_1, ..., \mathtt{v}_{i-1}, \mathtt{v}, \mathtt{v}_{i+1}, ..., \mathtt{v}_n]\}$$

The above axioms illustrate the main characteristics of separation logic: in order to analyse a heap-accessing command, it must be explicitly proved that the heap location under consideration is allocated. Meanwhile, the reward is that any other heap locations can be ignored safely.

The interplay of separation conjunction and tight interpretation makes local reasoning possible, which is formalised by the frame rule in separation logic:

$$\frac{\{\Delta_1\}\ e\ \{\Delta_2\}}{\{\Delta_1 * \Delta_3\}\ e\ \{\Delta_2 * \Delta_3\}}\ \mathsf{mods}(e) \cap \mathsf{fv}(\Delta_3) = \emptyset$$

where $\mathsf{mods}(e)$ returns the set of variables modified by command e. Note that $\mathsf{mods}(e)$ includes neither modified fields, nor the variables used to reach these fields. $\mathsf{fv}(\Delta_3)$ returns the set of free variables occurring in formula Δ_3. The crucial power of the frame rule is that it allows a global property to be derived from a local one, without necessity to look at other parts of the program.

3.3 Specification Language

Our specification language is on the basis of a predicate-based specification methodology, wherein the main annotation construct is the shape predicate, each of which describes a data structure. Our aim of using this scheme is to allow users to design their own predicates for shapes and relevant properties (numerical and content ones), to capture the desired level of program correctness to be verified. The advantages of this methodology include that it unifies heterogeneous techniques and annotations in a homogeneous way for the verification of linked data structures. Predicates also eliminate the need for an explicit ownership scheme; they capture sufficient information for us to perform verification of properties that involve closures. Finally, it permits us to easily decompose the properties to be verified for a shape predicate, which is beneficial for our aim of various levels of program correctness.

Before concrete examples of our shape predicates given in the next section, we introduce the grammar for the specification language in Figure 3.3. Each shape predicate *spred* has a name c, a list of parameters $\boldsymbol{v}$, and a body Φ. Each predicate also has a parameter `root`, written to the left of the predicate name c, which denotes a `root` pointer to the data structure captured by the predicate. A `root` pointer is one from which all objects in the data structure can be reached. `root` is a reserved identifier used only in predicate definitions. Φ is a normalised state which is essentially a separation logic formula in disjunctive normal form. The procedure specifications *mspec* are written in these states where Φ_{pr} and Φ_{po} denote the precondition and postcondition, respectively. Each disjunct σ consists of a heap formula κ and a pure formula π. The heap formula κ consists of $*$-conjoined atomic heap formulae $p{::}c\langle\boldsymbol{v}\rangle$. Such atomic heap formula $p{::}c\langle\boldsymbol{v}\rangle$ can denote either (i) a points-to fact $p \mapsto c[\boldsymbol{v}]$ if c is a class name, or (ii) a predicate instance $c(p, \boldsymbol{v})$ if c is a predicate name. The pure part π consists of heap-independent formulae, such as formulae for Presburger arith-

metic, formulae for pointer equality/disequality and formulae in multiset theory. As shown in the figure, Presburger arithmetic formulae (s) is made up of integer constraints, variables, addition, subtraction, scalar multiplication, maximum/minimum values and cardinality of multiset. For multiset (bag) theory, we allow expression of (quantified) value membership, subset relationship and bag arithmetic (such as union, intersection and subtraction). To make automated verification possible, we require that there is a sound and terminating procedure to decide the validity of heap-independent logic.

For the verification of programs, we regard σ as a *conjunctive abstract program state*, and use SH to denote a set of such conjunctive states. During a verification process, the *abstract program state* at each program point will be a disjunction of σ's denoted as Δ, and we name the set of such formulae as $\mathcal{P}_{\mathsf{SH}}$. Note that constraint abstractions ($\mathtt{Q}(\boldsymbol{v})$) may occur in Δ during the analysis. A closed-form Δ (containing no constraint abstractions) can be normalised to the Φ form (Nguyen et al., 2007). Meanwhile, we also have counterparts of σ and Δ in the pure (heap-independent) domain, say ω and Υ. We distinguish them here because later in our verification approach we will reduce the heap-relevant abstract states down to heap-independent pure states, and use existing provers to solve the pure constraints composed by these pure states.

Using entailment provided by the Sleek prover (Nguyen et al., 2007), we define a partial order over the abstract states

$$\Delta \preceq \Delta' =_{df} \Delta \vdash \Delta' * \mathtt{true}$$

A last notation to be described here, *lemma*, represents the lemmas defined in the program. They provide ways to soundly coerce predicates beyond their original definitions, and to specify the entailment relationship among predicates. For instance,

we can view a sorted list as a normal list as well, and our entailment prover will be capable of reasoning that a sorted list implies a normal list. Concrete examples of lemmas are also given in the subsequent sections.

Finally, when we write abstract program states or program specifications, we use three kinds of variables: program variables, logical variables related to program variables' shapes (such as a list's length), and logical variables to record intermediate states. For the first two groups we use variables without subscription (such as `x` and `xn`), and denote a program variable's initial value as unprimed, and its current (and hence latest) value as primed (Nguyen et al., 2007; Chin et al., 2007). For the third group, we use subscript ones like $\texttt{x}_1$ and $\texttt{xn}_1$. For instance, for a code segment $\texttt{x} = \texttt{x} + 1;\ \texttt{x} = \texttt{x} - 2$ starting with state $\{\texttt{x}{>}1\}$, we have the following reasoning procedure:

$$\{\texttt{x}'{=}\texttt{x} \wedge \texttt{x}{>}1\}\ \texttt{x=x+1}\ \{\texttt{x}{>}1 \wedge \texttt{x}'{=}\texttt{x}{+}1\}\ \texttt{x=x-2}\ \{\texttt{x}{>}1 \wedge \texttt{x}'{=}\texttt{x}_1{-}2 \wedge \texttt{x}_1{=}\texttt{x}{+}1\}$$

where the final value of `x` is recorded in variable $\texttt{x}'$ and $\texttt{x}_1$ keeps an intermediate state of `x`.

3.3.1 Shape Predicates and Lemmas

Our specification language allows user to describe both the shape of data structures as well as their quantitative properties and contents, and to use them to capture the desired level of program correctness. Shape invariants of the data structures are described using separation logic. Quantitative invariants, such as numerical properties and content of collections, are described using arithmetic or multiset formulae. For example, with a singly-linked list node

```
class Node { int val; Node next; }
```

Shape predicate	*spred*	$::= \texttt{root}::c\langle \boldsymbol{v} \rangle \equiv \Phi$
Lemma	*lemma*	$::= \texttt{root}::c\langle \boldsymbol{v} \rangle \wedge \pi \longleftarrow \Phi$
Specification	*mspec*	$::= \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po}$
Abstract state	Δ	$::= \texttt{Q}(\boldsymbol{v}) \mid \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta$
Normalised state	Φ	$::= \bigvee \boldsymbol{\sigma}$
Conjunctive state	σ	$::= \exists \boldsymbol{v} \cdot \kappa \wedge \pi$
Heap formula	κ	$::= \texttt{emp} \mid v::c\langle \boldsymbol{v} \rangle \mid \kappa_1 * \kappa_2$
Pure state	Υ	$::= \texttt{P}(\boldsymbol{v}) \mid \bigvee \boldsymbol{\omega} \mid \Upsilon_1 \wedge \Upsilon_2 \mid \Upsilon_1 \vee \Upsilon_2 \mid \exists v \cdot \Upsilon$
Pure conj. state	ω	$::= \exists \boldsymbol{v} \cdot \pi$
Pure formula	π	$::= \gamma \wedge \phi \mid \pi_1 \wedge \pi_2$
Aliasing	γ	$::= v_1 = v_2 \mid v = \texttt{null} \mid v_1 \neq v_2 \mid v \neq \texttt{null} \mid \gamma_1 \wedge \gamma_2$
Pure constr.	ϕ	$::= \varphi \mid b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg \phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi$
Boolean	b	$::= \texttt{true} \mid \texttt{false} \mid v \mid b_1 = b_2$
Numerical constr.	a	$::= s_1 = s_2 \mid s_1 \leq s_2$
Presburger arith.	s	$::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}} \times s \mid s_1 + s_2 \mid -s \mid \mathit{max}(s_1,s_2) \mid \mathit{min}(s_1,s_2) \mid \lvert \texttt{B} \rvert$
Bag constr.	φ	$::= v \in \texttt{B} \mid \texttt{B}_1 = \texttt{B}_2 \mid \texttt{B}_1 \sqsubset \texttt{B}_2 \mid \texttt{B}_1 \sqsubseteq \texttt{B}_2 \mid \forall v \in \texttt{B} \cdot \phi \mid \exists v \in \texttt{B} \cdot \phi$
Bag arith.	$\texttt{B}$	$::= \texttt{B}_1 \sqcup \texttt{B}_2 \mid \texttt{B}_1 \sqcap \texttt{B}_2 \mid \texttt{B}_1 - \texttt{B}_2 \mid \{\} \mid \{v\}$

Figure 3.3: The specification language.

as data structure, a user interested in pointer-safety may define a predicate to depict the list shape as in Distefano et al. (2006); Calcagno et al. (2009):

$$\texttt{root::list}\langle\rangle \;\equiv\; (\texttt{root=null}) \vee (\exists \texttt{i}, \texttt{q} \cdot \texttt{root::Node}\langle \texttt{i}, \texttt{q}\rangle * \texttt{q::list}\langle\rangle)$$

The sole parameter `root` for the predicate `list` is the root pointer referring to the list. As mentioned earlier, we use a uniform notation $\texttt{p::c}\langle \boldsymbol{v}\rangle$ to denote either a singleton heap or a predicate. If `c` is a class type node, the notation represents a singleton heap, $\texttt{p}\mapsto\texttt{c}[\boldsymbol{v}]$, e.g. the $\texttt{root::Node}\langle \texttt{i}, \texttt{q}\rangle$ above. If `c` is a predicate name, then the data structure pointed to by `p` has the shape `c` with parameters $\boldsymbol{v}$, e.g., the $\texttt{q::list}\langle\rangle$ above. In the inductive case, the separation conjunction $*$ ensures that two heap portions (representing respectively the head node and the tail list) are domain-disjoint. Our predicates use existential quantifiers for local values and pointers, such as `i` and `q`.

Yet another user may be interested to track also the length of a list to analyse quantitative measures, such as heap/stack resource usage. Therefore the predicate can be defined in a similar manner as in Magill et al. (2008):

$$\texttt{ll}\langle\texttt{n}\rangle \equiv (\texttt{root=null} \wedge \texttt{n=0}) \vee (\texttt{root::Node}\langle \_, \texttt{q}\rangle * \texttt{q::ll}\langle\texttt{m}\rangle \wedge \texttt{n=m+1})$$

where we use the following shortened notation: (i) default `root` parameter in LHS may be omitted, (ii) unbound variables, such as `q` and `m`, are implicitly existentially quantified, and (iii) _ denotes existentially quantified anonymous variable. The parameter `n` of the predicate represents an abstract value. Such value is not taken from a concrete heap location, but rather is computed from the pure formulae, which are usually based on the structure of the underlying heap. During a verification, this value is derived automatically by entailment, when a predicate is proved from a program state.

Meanwhile, this predicate may still be extended to support a higher-level of correct-

ness with multiset (bag) property to capture the list's content:

$$\texttt{llB}\langle \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\emptyset) \vee (\texttt{root}{::}\texttt{Node}\langle \texttt{v}, \texttt{q}\rangle * \texttt{q}{::}\texttt{llB}\langle \texttt{S}_1\rangle \wedge \texttt{S}{=}\{\texttt{v}\}{\sqcup}\texttt{S}_1)$$

which also implicitly suggests the list's length with $|\texttt{S}|$. This predicate can be strengthened furthermore if necessary, so as to verify a sorting algorithm:

$$\begin{aligned}\texttt{sllB}\langle \texttt{S}\rangle \equiv\ & (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\emptyset) \vee \\ & (\texttt{root}{::}\texttt{Node}\langle \texttt{v}, \texttt{q}\rangle * \texttt{q}{::}\texttt{sllB}\langle \texttt{S}_1\rangle \wedge \texttt{S}{=}\{\texttt{v}\}{\sqcup}\texttt{S}_1 \wedge (\forall \texttt{x}{\in}\texttt{S}_1{\cdot}\texttt{v}{\leq}\texttt{x}))\end{aligned}$$

The constraint $\forall \texttt{x}{\in}\texttt{S}_1{\cdot}\texttt{v}{\leq}\texttt{x}$ guarantees the sortedness property is adhered in the predicate. Therefore, it can be seen that the user is expected to provide predicate definitions in terms of their required correctness level and program properties. These predicates may be non-trivial but can be reused multiple times for specifications of different methods. We have also built a library of predicates with respect to commonly-used data structures and useful program properties.

One more note about the predicates is that we allow users to write lemmas to express their coercion relationship, which informally means some ways to observe a predicate other than its original definition. For example, if we define a predicate for a list segment with length `n` as

$$\texttt{ls}\langle \texttt{p}, \texttt{n}\rangle \equiv (\texttt{root}{=}\texttt{p} \wedge \texttt{n}{=}0) \vee (\texttt{root}{::}\texttt{Node}\langle \_, \texttt{q}\rangle * \texttt{q}{::}\texttt{ls}\langle \texttt{p}, \texttt{m}\rangle \wedge \texttt{n}{=}\texttt{m}{+}1)$$

where `p` represents the `next` field of the list's tail (last node). In this case, we may use lemma to present one more way to view a list as

$$\texttt{root}{::}\texttt{ll}\langle \texttt{n}\rangle \wedge \texttt{n}{=}\texttt{m}{+}\texttt{k} \longleftarrow \texttt{root}{::}\texttt{ls}\langle \texttt{p}, \texttt{m}\rangle * \texttt{p}{::}\texttt{ll}\langle \texttt{k}\rangle$$

which says a list of length `n` can be combined from two parts, one being a list segment with `m` nodes ending at `p` and the other a list with length `k` pointed to by `p`, with an additional constraint $\texttt{n}{=}\texttt{m}{+}\texttt{k}$. Also we can write a similar lemma to express that a list segment can be formed in an analogous way from two shorter list segments:

$$\texttt{root}{::}\texttt{ls}\langle \texttt{q}, \texttt{n}\rangle \wedge \texttt{n}{=}\texttt{m}{+}\texttt{k} \longleftarrow \texttt{root}{::}\texttt{ls}\langle \texttt{p}, \texttt{m}\rangle * \texttt{p}{::}\texttt{ls}\langle \texttt{q}, \texttt{k}\rangle$$

Such lemma is beneficial for our abstraction, abduction and specification synthesis procedures introduced in later chapters, since it provides another perspective to view the predicates other than their original definition. Therefore it offers more opportunity for the techniques above to succeed.

3.3.2 Well-Formedness and Well-Foundedness

To ensure the soundness and termination of our reasoning procedures, as in Nguyen et al. (2007), we require the shape predicates and specifications written with them be well-formed. To define this concept we first need to clarify the reachability of a heap constraint node from a variable:

Definition 3.3.1 (Reachability) *Given a heap constraint* $\kappa = p{::}c\langle \boldsymbol{v} \rangle * \kappa_1$, *node* $p{::}c\langle \boldsymbol{v} \rangle$ *is* reachable *from a variable* q *if and only if the following recursively defined relation holds:*

$$\begin{aligned} \texttt{reach}(\kappa, q, p{::}c\langle \boldsymbol{v} \rangle) \;=_{df}\; & (p = q) \vee \\ & (\kappa_1 = q{::}c_q\langle \ldots, r, \ldots \rangle * \kappa_2 \wedge \texttt{reach}(\kappa_2, r, p{::}c\langle \boldsymbol{v} \rangle)) \end{aligned}$$

On the basis of such reachability we define the well-formedness of predicate:

Definition 3.3.2 (Well-Formed Predicate) *A shape predicate* $\texttt{root}{::}c\langle \boldsymbol{v} \rangle \equiv \Phi$ *is well-formed if (i) every class node and shape predicate in* Φ *is reachable from either the parameters* $\boldsymbol{v}$ *or one of the special variables (*`root`/`res`*), (ii)* Φ *is in a disjunctive normal form* $\bigvee (\exists \boldsymbol{v} \cdot \boldsymbol{\kappa} \wedge \boldsymbol{\gamma} \wedge \boldsymbol{\phi})$ *where* κ *is for heap nodes,* γ *is for pointer constraints, and* ϕ *is for arithmetic and multiset formulae.*

The *well-formed* condition is significant in the light that all heap nodes of a heap formula must be reachable from the parameters and/or special variables. When

our entailment checking procedure (SLEEK) checks an entailment relationship, this condition allows it to correctly match nodes from the consequence with nodes from the antecedent.

Another potential problem during the reasoning is that arbitrary recursive shape relation can lead to non-termination entailment checking. To avoid that problem, we propose to use only *well-founded* shape predicates in our framework:

Definition 3.3.3 (Well-Founded Predicate) *A shape predicate is said to be* well-founded *if it satisfies four conditions, namely: (i) it is a well-formed predicate, (ii) the parameter* `root` *may only be bound to a class node and not a predicate, (iii) only* `root` *is allowed to be bound to a class node and (iv) every predicate is reachable from* `root`.

The shape predicates given in the last section are all well-founded. In contrast, the following three shape definitions are *not* well-founded (Nguyen et al., 2007):

$$\begin{array}{lcl} \texttt{foo}\langle\texttt{n}\rangle & \equiv & \texttt{root::foo}\langle\texttt{m}\rangle \wedge \texttt{n=m+1} \\ \texttt{goo}\langle\rangle & \equiv & \texttt{root::Node}\langle\_,\_\rangle * \texttt{q::goo}\langle\rangle \\ \texttt{too}\langle\rangle & \equiv & \texttt{root::Node}\langle\_,\texttt{q}\rangle * \texttt{q::Node}\langle\_,\_\rangle \end{array}$$

For `foo`, the `root` identifier is bound to a shape predicate. For `goo`, the heap node pointed by `q` is *not* reachable from variable `root` (therefore it is even not well-formed). For `too`, an extra object node is bound to a non-`root` variable. The first example may cause non-termination of entailment proof: when we want to rearrange a heap part of `foo` to expose an object from it we simply get another `foo` which requires another unfolding leading to non-termination. The second example captures an unreachable (junk) heap that cannot be located by our entailment procedure. The last example is just a syntactic restriction to facilitate termination proof of entailment checking, and can be easily overcome by introducing intermediate predicates.

3.3.3 Precondition and Postcondition

We allow procedures to be annotated with preconditions and postconditions. A precondition is an assertion that should be satisfied when a procedure is called, therefore the procedure body can assume it when the procedure starts. A postcondition is an assertion that should be established when the procedure exits, therefore the caller can assume it after the call, if the procedure is successfully verified. According to separation logic semantics, a precondition furthermore guarantees the existence of all memory locations that the procedure accesses, and thus guarantees executions free of memory errors.

For example, using the `llB` and `sllB` predicates, we can specify insertion sort algorithm operating on linked lists. The algorithm recursively sorts the tail of the input list, and inserts the first element into a sorted list such that order is maintained. Its code is in Figure 3.4.

From the code we can see the `insert_sort` procedure sorts a singly-linked list. As its precondition $\texttt{x}::\texttt{llB}\langle\texttt{S}\rangle \wedge |\texttt{S}|{\geq}1$ suggests, it takes in an unsorted list starting from `x` with content `S`, whose size should be at least one (this constraint is equivalent to $\texttt{x}{\neq}\texttt{null}$ and $\texttt{S}{\neq}\emptyset$, which, if the user specifies, can be captured by our entailment checker). Upon successful return it gives a sorted list with the same content, as captured by the postcondition $\texttt{res}::\texttt{sllB}\langle\texttt{T}\rangle \wedge \texttt{S}{=}\texttt{T}$.

The procedure `insert` inserts an object pointed to by `x` into a sorted list referenced by `r`. The separation conjunction $*$ constrains the object `x` not to belong to the list `r`, thereby the resulting list has one more element. Meanwhile, the returned pointer `res` points to a sorted list whose content is the union of the two inputs', as the postcondition indicates.

```
 1 class Node { int val; Node next; }
 2 Node insert_sort(Node x)
 3  requires x::llB⟨S⟩ ∧ |S|≥1
 4  ensures  res::sllB⟨T⟩ ∧ S=T {
 5   if (x.next == null) return x;
 6   else { Node s = x.next;
 7     Node r = insert_sort(s);
 8     return insert(r, x);
 9   }
10 }
11 Node insert(Node r, Node x)
12  requires r::sllB⟨S⟩ * x::Node⟨v, _⟩
13  ensures  res::sllB⟨T⟩ ∧ T=S⊔{v} {
14   if (r == null) {
15     x.next = null; return x;
16   } else if (x.val <= r.val) {
17     x.next = r; return x;
18   } else {
19     r.next = insert(r.next, x);
20     return r;
21   }
22 }
```

Figure 3.4: The insertion sort program for singly linked lists.

3.3.4 The Semantic Model

The semantics of our specification formulae is adapted from what is given for the "early versions" of separation logic (Ishtiaq and O'Hearn, 2001; Reynolds, 2002),

except that we have extensions to handle user-defined shape predicates and related pure properties. We assume sets Loc of memory locations, Val of primitive values (with $0 \in \mathsf{Val}$ denoting `null`), Var of variables (program and logical variables), and ObjVal of object values stored in the heap, with $c[f_1 \mapsto \nu_1, \ldots, f_n \mapsto \nu_n]$ denoting an object value of class c where $\nu_1, \ldots, \nu_n$ are current values of the corresponding fields $f_1, \ldots, f_n$. Let $s, h \models \Delta$ denote the model relation, i.e. the stack s and heap h satisfy Δ, with h, s from the following concrete domains:

$$\begin{array}{lll} h \in \mathit{Heaps} & =_{df} & \mathsf{Loc} \rightharpoonup_{\mathit{fin}} \mathsf{ObjVal} \\ s \in \mathit{Stacks} & =_{df} & \mathsf{Var} \rightarrow \mathsf{Val} \cup \mathsf{Loc} \end{array}$$

Note that each heap h is a finite partial mapping while each stack s is a total mapping, as in the classical separation logic (Ishtiaq and O'Hearn, 2001; Reynolds, 2002). The detailed model definition is in Figure 3.5.

$$\begin{array}{lll} s, h \models \Phi_1 \vee \Phi_2 & \texttt{iff} & s, h \models \Phi_1 \texttt{ or } s, h \models \Phi_2 \\ s, h \models \exists \boldsymbol{v} \cdot \kappa \wedge \pi & \texttt{iff} & \exists \boldsymbol{\nu} \cdot s[\boldsymbol{v} \mapsto \boldsymbol{\nu}], h \models \kappa \texttt{ and } s[\boldsymbol{v} \mapsto \boldsymbol{\nu}] \models \pi \\ s, h \models \kappa_1 * \kappa_2 & \texttt{iff} & \exists h_1, h_2 \cdot h_1 \perp h_2 \texttt{ and } h = h_1 \cdot h_2 \texttt{ and} \\ & & s, h_1 \models \kappa_1 \texttt{ and } s, h_2 \models \kappa_2 \\ s, h \models \texttt{emp} & \texttt{iff} & \mathsf{dom}(h) = \emptyset \\ s, h \models p{::}c\langle v_1, \ldots, v_n \rangle & \texttt{iff} & \mathit{IsObj}(c) \texttt{ and } s(p) \in \mathsf{Loc} \texttt{ and } h = [s(p) \mapsto r] \texttt{ and} \\ & & r = c[f_1 \mapsto s(v_1), \ldots, f_n \mapsto s(v_n)] \\ & \texttt{or} & \mathit{IsPred}(c) \texttt{ and } s, h \models [p/\texttt{root}]\Phi \\ s \models \pi_1 \wedge \pi_2 & \texttt{iff} & s \models \pi_1 \texttt{ and } s \models \pi_2 \\ s \models \pi & \texttt{iff} & s \models_A \pi \end{array}$$

Figure 3.5: The semantic model.

We do not provide a semantics for the placeholder `Q` of constraint abstraction here, as its meaning is described by the formula it represents. For pure formulae π, as noted

in the last line of Figure 3.5, their semantics are defined with a specific notation $\models_A$, which is preserved by the pure constraint provers that we use for soundness purpose. Its definition is given in Figure 3.6.

$s \models_A \gamma_1 \wedge \gamma_2$	`iff`	$s \models_A \gamma_1$ `and` $s \models_A \gamma_2$
$s \models_A p_1 \bowtie p_2$	`iff`	$s(p_1) \bowtie s(p_2)$, `where` $\bowtie \in \{=, \neq\}$
$s \models_A p \bowtie$ `null`	`iff`	$s(p) \bowtie 0$, `where` $\bowtie \in \{=, \neq\}$
$s \models_A$ `true`	`always`	
$s \models_A$ `false`	`never`	
$s \models_A v$	`iff`	$s(v) =$ `true`
$s \models_A b_1 = b_2$	`iff`	$s(b_1) = s(b_2)$
$s \models_A v_1 = v_2$	`iff`	$s(v_1) = s(v_2)$
$s \models_A v_1 \leq v_2$	`iff`	$s(v_1) \leq s(v_2)$
$s \models_A \phi_1 \wedge \phi_2$	`iff`	$s \models_A \phi_1$ `and` $s \models_A \phi_2$
$s \models_A \phi_1 \vee \phi_2$	`iff`	$s \models_A \phi_1$ `or` $s \models_A \phi_2$
$s \models_A \neg\phi$	`iff`	$s \models_A \phi$ `does not hold`
$s \models_A \exists v \cdot \phi$	`iff`	$s \models_A [k/v]\phi$ `for some` k
$s \models_A \forall v \cdot \phi$	`iff`	$s \models_A [k/v]\phi$ `for all` k
$s \models_A v \in \mathtt{B}$	`iff`	$s(v) \in s(\mathtt{B})$
$s \models_A \mathtt{B}_1 = \mathtt{B}_2$	`iff`	$s(\mathtt{B}_1) = s(\mathtt{B}_2)$
$s \models_A \mathtt{B}_1 \sqsubset \mathtt{B}_2$	`iff`	$s(\mathtt{B}_1) \subset s(\mathtt{B}_2)$
$s \models_A \mathtt{B}_1 \sqsubseteq \mathtt{B}_2$	`iff`	$s(\mathtt{B}_1) \subseteq s(\mathtt{B}_2)$
$s \models_A \forall v \in \mathtt{B} \cdot \phi$	`iff`	$s \models_A [k/v]\phi$ `for all` $k \in s(\mathtt{B})$
$s \models_A \exists v \in \mathtt{B} \cdot \phi$	`iff`	$s \models_A [k/v]\phi$ `for some` $k \in s(\mathtt{B})$

Figure 3.6: The semantic model for pure constraints.

3.4 Summary

This chapter defines the programming language as our verification target as well as the specification language to depict program contracts, where the latter is founded on the basis of separation logic (Reynolds, 2002). For the purpose to prove our approach's soundness, we introduce the operational semantics of the programming language and the semantic model for the specification language. Meanwhile, we also illustrate the language settings with several informal examples.

Chapter 4

Refining Partial Specifications for Verification

Automatically verifying the functional correctness of heap-manipulating programs with complex data structures is a challenging task. This process can greatly benefit from human assistance through specification annotations. However, it requires much intellectual effort from users, and meanwhile users are liable to make mistakes in writing such specifications. In this chapter, we propose a new approach to program verification that allows users to provide only partial specifications to methods. Our approach will then refine the given annotations into more complete specifications by discovering missing constraints. The discovered constraints may involve both numerical and multiset properties that could be later confirmed or revised by users. Therefore, with our approach, we are able to increase the level of verification automation and save users' effort.

4.1 Introduction

As discussed in previous chapters, the research on software verification has a long and distinguished history dating back to the 1960's. Nevertheless, it remains a challenging problem to automatically verify heap-manipulating programs written in mainstream imperative languages. This is in part due to the shared mutable data structures lying in programs, and the need to track related "pure" properties, such as structural numerical information (size and height), relational numerical information (balanced and sortedness properties), and content information (multiset of symbolic values). These properties, intertwined with each other, are non-trivial to reason about in a precise and concise way. The crux is that they span over several abstract domains (shape, numerical and multiset), each of which has infinite state space and various reasoning rules, resulting in an even larger combined state space and hence making the whole verification work exceptionally complicated.

Human assistance is often essential in (semi-) automated program verification. The user may supply annotations at certain program point, such as loop invariants and/or method specifications. These annotations can greatly narrow down the possible program states at that point, and avoid fixed-point calculation which could be expensive and may be less precise than the user's insight.

However, an obvious disadvantage of user annotation concerns its scalability, since programs to be verified may be complicated and their functions are also diverse. Accordingly their specifications are difficult to compose, especially for a complex program of many modules and methods. Therefore, it is not preferable to require the user to provide specification for each method and invariant for each loop when verifying a relatively large software system, as such workload would counteract the guarantee of program correctness brought by verification. Meanwhile, human is

liable to make mistakes. A programmer may under-specify with too weak a precondition or over-specify with too strong a postcondition. Such mistakes could lead to failed verification, and it may be difficult for the user to discover whether the error is due to a real bug in the program, or an inappropriately supplied annotation.

To balance verification quality and human effort, we provide a novel approach to the verification of heap-manipulating programs. Under our framework, the user is expected to provide *partial* specifications for programs with only *shape* information. Our verification will then take over the rest of the work to refine those partial specifications with derived (pure) constraints which should be satisfied by the program, or report a possible program bug if the given specifications are rejected by our verifier. This is more beneficial compared with previous works (Nguyen et al., 2007; Chin et al., 2007; Nguyen and Chin, 2008), where users must provide full specifications to verify programs, including not only shape information but also pure properties such as size and multiset.

As introduced in Chapter 3, we allow users to design their own predicates and use such predicates to capture their required correctness level and program properties. For example, with the data structure `class Node { int val; Node next; }` defined in the last chapter, we may have four different predicates to express various program properties according to our demand: `list`, `ll`, `llB` and `sllB` (page 50). Based on these predicates, the user is expected to provide partial specifications for procedures in programs. Say, for the main procedure of the insertion sort algorithm in the previous chapter (page 57) taking `x` as an input parameter that is expected to be non-null, the user may provide $\texttt{x::llB}\langle\texttt{S}\rangle$ as precondition and $\texttt{x::sllB}\langle\texttt{T}\rangle$ as postcondition, and our approach will refine the specification as $\texttt{x::llB}\langle\texttt{S}\rangle \wedge |\texttt{S}|{\geq}1$ for pre, and $\texttt{x::sllB}\langle\texttt{T}\rangle \wedge \texttt{S}{=}\texttt{T}$ for post. Here we need the user annotations as initial specification, because we reserve the flexibility of verification with respect to different program properties at various correctness levels. For example, our approach can also verify

the same algorithm, but for the following refined specifications:

$$\begin{array}{llll}
\texttt{requires} & \texttt{x::list}\langle\rangle \wedge \texttt{x}\neq\texttt{null} & \texttt{ensures} & \texttt{x::list}\langle\rangle \\
\texttt{requires} & \texttt{x::ll}\langle\texttt{m}\rangle \wedge \texttt{m}>0 & \texttt{ensures} & \texttt{x::ll}\langle\texttt{n}\rangle \wedge \texttt{m}=\texttt{n} \\
\texttt{requires} & \texttt{x::llB}\langle\texttt{S}\rangle \wedge |\texttt{S}|\geq 1 & \texttt{ensures} & \texttt{x::llB}\langle\texttt{T}\rangle \wedge \texttt{S}=\texttt{T} \\
\texttt{requires} & \texttt{x::llB}\langle\texttt{S}\rangle \wedge |\texttt{S}|\geq 1 & \texttt{ensures} & \texttt{x::ll}\langle\texttt{n}\rangle \wedge |\texttt{S}|=\texttt{n}
\end{array}$$

where the discovered missing constraints are shown in shaded form. This can help reduce the number of redundant specifications considered.

To summarise, our proposal for refining partial specification is aimed at harnessing the synergy between human's insights and machine's capability at automated program analysis. In particular, human's guidance can help narrow down on the most important of the numerous specifications that are possible with each program code, while automation by machine is important for minimising on the tedium faced by users, and to support easier adoption of automated verification technology. Our proposal has the following characteristics:

- *Specification completion*: This verification refines the specification from three aspects, namely, the constraints needed in the precondition for memory and code safety, the constraints in postcondition to link the method's pre- and post-states, and the constraints that the method's post-state satisfies.
- *Flexibility*: We allow the user to define their own predicates for the program properties they want to verify, so as to provide different levels of correctness. Meanwhile we aim at, and have covered much of, both memory safety and functional correctness of pointer-manipulating programs such as data structure shapes, pointer safety, structural/relational numerical constraints, and multiset information.
- *Reduction of user annotations*: Our approach uses program analysis techniques effectively to reduce users' annotations, as will be exhibited by our experiments

in Chapter 7.

The remainder of this chapter is organised as follows. We will first depict our approach informally using a motivating example in Section 4.2, and present technical details thereafter in Section 4.3. More related works and concluding remarks come at last.

4.2 The Approach

In this section, we use two motivating examples to informally illustrate our approach. The first example is about the insertion sort in the previous chapter, while the second example is more sophisticated involving both lists and trees, and a transformation between their shapes, which witnesses our approach's enhanced capability compared with related works.

4.2.1 An Illustrative Example

We illustrate our approach using method `insert_sort` in Figure 4.1. We show how our verification infers missing constraints to improve the user-supplied incomplete specification.

The program code of `insert_sort` and `insert` are exactly the same as that in Figure 3.4. Their difference resides in the annotations, as the one in Figure 4.1 does not have pure constraints (quantitative or content ones). As may be observed, the main procedure's specification just addresses that the algorithm will transfer an unsorted list into a sorted one, yet without any further obligations over the length of

```
 1 class Node { int val; Node next; }
 2 Node insert_sort(Node x)
 3  requires x::llB⟨S⟩
 4  ensures  res::sllB⟨T⟩ {
 5   if (x.next == null) return x;
 6   else { Node s = x.next;
 7     Node r = insert_sort(s);
 8     return insert(r, x);
 9   }
10 }
11 Node insert(Node r, Node x)
12  requires r::sllB⟨S⟩ ∗ x::Node⟨v, _⟩
13  ensures  res::sllB⟨T⟩ {
14   if (r == null) {
15     x.next = null; return x;
16   } else if (x.val <= r.val) {
17     x.next = r; return x;
18   } else {
19     r.next = insert(r.next, x);
20     return r;
21   }
22 }
```

Figure 4.1: The insertion sort program for singly linked lists.

the list or its content. This has two problems: one is unsoundness and the other is loss of precision. Here the second problem suggests that the specification loses the information that the content of the transferred list should be identical as the input, which is crucial for the functional correctness proof of the algorithm but missing in the postcondition. The first problem is more severe as the given precondition even does not guarantee memory safety of the program execution (when the input list is empty the program will fail). Meanwhile, the second problem also applies to `insert`'s partial specifications. Therefore we need to infer such constraints in order that the specifications become both sound and precise.

To verify `insert_sort`, our approach proceeds in two steps. Firstly, starting from the partial precondition, a forward analysis is conducted to compute the postcondition of the method in the form of a constraint abstraction, as mentioned in Chapter 2. This constraint abstraction is effectively a transfer function for the method, which may be recursively defined. During this analysis, abductive reasoning may be used whenever the current state fails to establish the precondition of the next program command. Secondly, instead of a direct fixed-point computation in the combined abstract domain (with shape, numerical and multiset information), a "pure" constraint abstraction (without heap shape information) is derived from the generated constraint abstraction and the user-given partial postcondition. This pure constraint abstraction is then solved by fixed-point solvers in pure (numerical/multiset) domains, such as Nipkow et al. (2002); Popeea and Chin (2006).

The constraint abstraction of a code segment (say, a method) in our settings is an abstraction form of that code's postcondition, given a certain precondition. As the code may contain loops or recursive calls, its constraint abstraction can also be recursive, or in an *open form*, accordingly. To illustrate, for the following while loop

$$\texttt{while}\ (\texttt{x} > 0)\ \{\, \texttt{x} = \texttt{x} - 1;\ \texttt{y} = \texttt{y} + 2; \}$$

and its precondition

$$\mathtt{x}{\geq}0 \wedge \mathtt{y}{=}0$$

we have its constraint abstraction as

$$\mathtt{Q}(\mathtt{x},\mathtt{x}',\mathtt{y},\mathtt{y}') ::= \mathtt{x}{\leq}0 \wedge \mathtt{x}{=}\mathtt{x}' \wedge \mathtt{y}{=}\mathtt{y}' \ \vee \ \mathtt{x}{>}0 \wedge \mathtt{Q}(\mathtt{x}{-}1,\mathtt{x}',\mathtt{y}{+}2,\mathtt{y}')$$

where we denote $\mathtt{x}$ and $\mathtt{y}$ as their values before the loop, and the primed versions as their current values. Such constraint abstraction presents the invariant of the while loop. Its fixed-point can normally be achieved with a standard fixed-point calculation process, with result $2(\mathtt{x}{-}\mathtt{x}') = \mathtt{y}' - \mathtt{y}$. However, such fixed-point calculation is generally in the pure domain at present, whereas our constraint abstraction should be more complicated involving both shape and pure constraints, requiring us to split them for solution somehow.

As for the example, our forward analysis runs on the body of `insert_sort` to construct the constraint abstraction. For lines 5-9, it produces a disjunction as the effect of if-else (according to the if-else rule in page 90):

$$\mathtt{Q}(\mathtt{x},\mathtt{S},\mathtt{res},\mathtt{T}) ::= (\text{post-state of if}) \vee (\text{post-state of else})$$

where $\mathtt{Q}$ represents the post-state of the if-else statement (as well as the method), and its parameters $\mathtt{x},\mathtt{S},\mathtt{res}$ and $\mathtt{T}$ are the (program and logical) variables involved in the state.

For the if branch, after the unfolding over $\mathtt{x}{::}\mathtt{llB}\langle\mathtt{S}\rangle$ (rule unfold in page 86), we know from the condition that the input sorted list $\mathtt{x}$ has only one node, and thus its post-state will be

$$\exists \mathtt{v} \cdot \mathtt{x}{::}\mathtt{Node}\langle\mathtt{v},\mathtt{null}\rangle \wedge \mathtt{res}{=}\mathtt{x} \wedge \mathtt{S}{=}\{\mathtt{v}\}$$

Meanwhile, for the else branch, the sorted list will firstly be unrolled by one node at line 6 (rule unfold), making $\mathtt{x.next}$ point to $\mathtt{s}$ (rule assign in page 90), which

references a sub-list one node shorter than the input list beginning from x:

$$\exists \mathtt{S_s}, \mathtt{v} \cdot \mathtt{x}{::}\mathtt{Node}\langle \mathtt{v}, \mathtt{s}\rangle * \mathtt{s}{::}\mathtt{llB}\langle \mathtt{S_s}\rangle \wedge \mathtt{S}{=}\mathtt{S_s}{\sqcup}\{\mathtt{v}\}$$

After that, insert_sort is invoked recursively with s. It will consume the precondition ($\mathtt{s}{::}\mathtt{llB}\langle \mathtt{S_s}\rangle$) and ensure the postcondition (in terms of Q, partially according to the rule in page 88; however it will be substituted as described later). In that case, the state immediately after symbolic execution of line 7 is

$$\mathtt{Q}(\mathtt{x}, \mathtt{S}, \mathtt{res}, \mathtt{T}) ::= \exists \mathtt{v} \cdot \mathtt{x}{::}\mathtt{Node}\langle \mathtt{v}, \mathtt{null}\rangle \wedge \mathtt{res}{=}\mathtt{x} \wedge \mathtt{S}{=}\{\mathtt{v}\} \vee$$
$$\exists \mathtt{v}, \mathtt{s}, \mathtt{S_s}, \mathtt{r}, \mathtt{S_r} \cdot \mathtt{x}{::}\mathtt{Node}\langle \mathtt{v}, \mathtt{s}\rangle * \mathtt{Q}(\mathtt{s}, \mathtt{S_s}, \mathtt{r}, \mathtt{S_r}) \wedge |\mathtt{S}|{>}1 \wedge \mathtt{S}{=}\mathtt{S_s}{\sqcup}\{\mathtt{v}\}$$

Here the first disjunctive branch corresponds to the base case in the method body, and the second branch captures the effect of the recursive call (with Q). Note that existential variables (not in the parameter list of Q) are local variables whose quantification may be omitted for brevity (as we will do so later).

Then the forward analysis continues over line 8 to invoke insert. Before the invocation we must ensure insert's precondition is satisfied. However, part of its requirement, the sorted list referenced by r, is within the instance of constraint abstraction $\mathtt{Q}(\mathtt{s}, \mathtt{S_s}, \mathtt{r}, \mathtt{S_r})$ in the second branch. For this purpose we replace $\mathtt{Q}(\mathtt{s}, \mathtt{S_s}, \mathtt{r}, \mathtt{S_r})$ with $\mathtt{r}{::}\mathtt{sllB}\langle \mathtt{S_r}\rangle \wedge \mathtt{P}(\mathtt{s}, \mathtt{S_s}, \mathtt{r}, \mathtt{S_r})$ to make explicit the heap portion referred to by r before we analyse the call $\mathtt{insert}(\mathtt{r}, \mathtt{x})$ (rule call-inf in page 88). This is safe because the following entailment relationship is added to our assumption:

$$\mathtt{Q}(\mathtt{x}, \mathtt{S}, \mathtt{res}, \mathtt{T}) \vdash \mathtt{res}{::}\mathtt{sllB}\langle \mathtt{T}\rangle \wedge \mathtt{P}(\mathtt{x}, \mathtt{S}, \mathtt{res}, \mathtt{T}) \tag{4.1}$$

which signifies that Q can be abstracted as a sorted list referenced by res plus some pure constraints P (also in constraint abstraction form, whose definition is to be derived in the next step). Based on this fact we may complete the replacement and invoke insert, whose specification can be obtained in a same manner as:[1]

$$\mathtt{requires}\ \mathtt{r}{::}\mathtt{sllB}\langle \mathtt{S}\rangle * \mathtt{x}{::}\mathtt{Node}\langle \mathtt{v}, \_\rangle\ \mathtt{ensures}\ \mathtt{res}{::}\mathtt{sllB}\langle \mathtt{T}\rangle \wedge \mathtt{T}{=}\mathtt{S}{\sqcup}\{\mathtt{v}\}$$

[1] We should apply the current approach over insert to refine its specifications with missing constraints beforehand. This process is introduced in Section 4.2.2.

which indicates that the returned sorted list has the same content as the input list (x) plus {v}. Applying it, we obtain the following post-state for insert_sort:

$$\texttt{Q}(\texttt{x},\texttt{S},\texttt{res},\texttt{T}) ::= \texttt{x::Node}\langle \texttt{v},\texttt{null}\rangle \wedge \texttt{res}{=}\texttt{x} \wedge \texttt{S}{=}\{\texttt{v}\} \vee$$
$$\texttt{res::sllB}\langle \texttt{S}_{\texttt{res}}\rangle \wedge \texttt{P}(\texttt{s},\texttt{S}_{\texttt{s}},\texttt{r},\texttt{S}_{\texttt{r}}) \wedge |\texttt{S}|{>}1 \wedge \texttt{S}{=}\texttt{S}_{\texttt{s}}{\sqcup}\{\texttt{v}\} \wedge \texttt{S}_{\texttt{res}}{=}\texttt{S}_{\texttt{r}}{\sqcup}\{\texttt{v}\}$$

The first disjunctive branch corresponds to the base case, and the second branch captures the effect of the recursive call as well as insert. In the base case, the method's return pointer (res) points to one node with value v. The recursive branch signifies that the post-state of the method concerns the recursive call and the call to insert (over s and r), as the constraint abstraction denotes. Note that T will be not available (as well as its relationship with $\texttt{S}_{\texttt{res}}$) until the next step.

In the second step, we first derive the definition of the pure constraint abstraction P from the above post-state Q. Each disjunctive branch of Q is used to entail the user-given post-shape (with appropriate instantiations of the parameters). The obtained frames form (via disjunction) the definition of P. For insert_sort, according to the entailment relationship (4.1), we obtain the following pure constraint abstraction:

$$\texttt{P}(\texttt{x},\texttt{S},\texttt{res},\texttt{T}) ::= (\texttt{T}{=}\texttt{S} \wedge |\texttt{S}|{=}1) \vee$$
$$(\texttt{P}(\texttt{s},\texttt{S}_{\texttt{s}},\texttt{r},\texttt{S}_{\texttt{r}}) \wedge |\texttt{S}|{>}1 \wedge \texttt{S}{=}\texttt{S}_{\texttt{s}}{\sqcup}\{\texttt{v}\} \wedge \texttt{T}{=}\texttt{S}_{\texttt{r}}{\sqcup}\{\texttt{v}\})$$

We then use pure fixed-point solvers (Nipkow et al., 2002; Popeea and Chin, 2006) to obtain a closed-form formula $|\texttt{S}|{\geq}1 \wedge \texttt{T}{=}\texttt{S}$ for P. On the basis of (4.1), we now obtain the closed-form approximation for Q:

$$\texttt{Q}(\texttt{x},\texttt{S},\texttt{res},\texttt{T}) ::= \texttt{res::sllB}\langle \texttt{T}\rangle \wedge |\texttt{S}|{\geq}1 \wedge \texttt{T}{=}\texttt{S}$$

The obtained pure formula is then used to refine the method's specification as

$$\texttt{requires x::llB}\langle \texttt{S}\rangle \wedge |\texttt{S}|{\geq}1 \quad \texttt{ensures res::sllB}\langle \texttt{T}\rangle \wedge \texttt{T}{=}\texttt{S}$$

which imposes more requirement in the precondition, stating that there should be at least one node in the list to be sorted for the sake of memory safety. With that obligation, the method guarantees that the result list is sorted and its content remains the same as the input list.

4.2.2 Refinement for the Specification of `insert`

The process to verify `insert` and refine its specification is analogous to the one for `insert_sort`. In the first step, we apply the forward analysis over `insert` to obtain the constraint abstraction representing the whole method as follows:

$$\begin{array}{l}\mathtt{Q}(\mathtt{r},\mathtt{S},\mathtt{x},\mathtt{v},\mathtt{res},\mathtt{T}) ::= \mathtt{x}::\mathtt{Node}\langle \mathtt{v},\mathtt{null}\rangle \wedge \mathtt{res}{=}\mathtt{x} \wedge \mathtt{r}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset \vee \\ \quad \mathtt{x}::\mathtt{Node}\langle \mathtt{v},\mathtt{r}\rangle * \mathtt{r}::\mathtt{sllB}\langle \mathtt{S}\rangle \wedge \mathtt{res}{=}\mathtt{x} \wedge (\forall \mathtt{u}{\in}\mathtt{S}{\cdot}\mathtt{v}{\leq}\mathtt{u}) \vee \\ \quad\quad \mathtt{r}::\mathtt{Node}\langle \mathtt{u},\mathtt{p}\rangle * \mathtt{Q}(\mathtt{q},\mathtt{S_q},\mathtt{x},\mathtt{v},\mathtt{p},\mathtt{T_p}) \wedge \mathtt{res}{=}\mathtt{r} \wedge \mathtt{u}{<}\mathtt{v}\end{array}$$

Then a replacement of the constraint abstraction instance with the post-shape is performed to yield

$$\begin{array}{l}\mathtt{Q}(\mathtt{r},\mathtt{S},\mathtt{x},\mathtt{v},\mathtt{res},\mathtt{T}) ::= \mathtt{x}::\mathtt{Node}\langle \mathtt{v},\mathtt{null}\rangle \wedge \mathtt{res}{=}\mathtt{x} \wedge \mathtt{r}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset \vee \\ \quad \mathtt{x}::\mathtt{Node}\langle \mathtt{v},\mathtt{r}\rangle * \mathtt{r}::\mathtt{sllB}\langle \mathtt{S}\rangle \wedge \mathtt{res}{=}\mathtt{x} \wedge (\forall \mathtt{u}{\in}\mathtt{S}{\cdot}\mathtt{v}{\leq}\mathtt{u}) \vee \\ \quad\quad \mathtt{r}::\mathtt{Node}\langle \mathtt{u},\mathtt{p}\rangle * \mathtt{p}::\mathtt{sllB}\langle \mathtt{T_p}\rangle \wedge \mathtt{P}(\mathtt{q},\mathtt{S_q},\mathtt{x},\mathtt{v},\mathtt{p},\mathtt{T_p}) \wedge \mathtt{res}{=}\mathtt{r} \wedge \mathtt{u}{<}\mathtt{v}\end{array}$$

Note that we do not have the relation $\mathtt{S}{=}\mathtt{S_q}{\sqcup}\{\mathtt{u}\}$ until we perform the entailment checking in the next step.

In the second step, applying the entailment relationship

$$\mathtt{Q}(\mathtt{r},\mathtt{S},\mathtt{x},\mathtt{v},\mathtt{res},\mathtt{T}) \vdash \mathtt{res}::\mathtt{sllB}\langle \mathtt{T}\rangle \wedge \mathtt{P}(\mathtt{r},\mathtt{S},\mathtt{x},\mathtt{v},\mathtt{res},\mathtt{T})$$

we reduce the constraint abstraction into the pure domain for solution:

$$\begin{array}{l}\mathtt{P}(\mathtt{r},\mathtt{S},\mathtt{x},\mathtt{v},\mathtt{res},\mathtt{T}) ::= \mathtt{res}{=}\mathtt{x} \wedge \mathtt{r}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset \wedge \mathtt{T}{=}\{\mathtt{v}\} \vee \\ \quad \mathtt{res}{=}\mathtt{x} \wedge \mathtt{T}{=}\mathtt{S}{\sqcup}\{\mathtt{v}\} \wedge (\forall \mathtt{u}{\in}\mathtt{S}{\cdot}\mathtt{v}{\leq}\mathtt{u}) \vee \\ \quad\quad \mathtt{P}(\mathtt{q},\mathtt{S_q},\mathtt{x},\mathtt{v},\mathtt{p},\mathtt{T_p}) \wedge \mathtt{res}{=}\mathtt{r} \wedge \mathtt{S}{=}\mathtt{S_q}{\sqcup}\{\mathtt{u}\} \wedge \mathtt{T}{=}\mathtt{T_p}{\sqcup}\{\mathtt{u}\} \wedge \mathtt{u}{<}\mathtt{v}\end{array}$$

which gives $\mathtt{T}{=}\mathtt{S}{\sqcup}\{\mathtt{v}\}$. Therefore we refine the original shape-only specification as

$$\mathtt{requires}\ \mathtt{r}::\mathtt{sllB}\langle \mathtt{S}\rangle * \mathtt{x}::\mathtt{Node}\langle \mathtt{v},\_\rangle\ \mathtt{ensures}\ \mathtt{res}::\mathtt{sllB}\langle \mathtt{T}\rangle \wedge \mathtt{T}{=}\mathtt{S}{\sqcup}\{\mathtt{v}\}$$

4.2.3 Another Illustrative Example

In this section we illustrate our approach with another more interesting example, which involves both linear data structures (lists) and non-linear ones (trees) as well as subtle pure properties.

Let us consider the method `sdl2nbt` shown in Figure 4.2. In the user-given (partial) specification, two predicates are used. The predicate below is used to represent sorted doubly-linked list segments:

$$\texttt{sdlB}\langle \texttt{p}, \texttt{q}, \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{q} \wedge \texttt{S}{=}\emptyset) \ \vee \ (\texttt{root}{::}\texttt{Node2}\langle \texttt{v}, \texttt{p}, \texttt{r}\rangle * \texttt{r}{::}\texttt{sdlB}\langle \texttt{root}, \texttt{q}, \texttt{S}_1\rangle \wedge \texttt{root}{\neq}\texttt{q} \wedge \texttt{S}{=}\{\texttt{v}\} \sqcup \texttt{S}_1 \wedge (\forall \texttt{x}{\in}\texttt{S}_1{\cdot}\texttt{v}{\leq}\texttt{x}))$$

where the parameters `p` and `q` denote the `prev` field of `root` and the `next` field of the list's last node, respectively. Meanwhile `S` is a multiset parameter to represent the list's content. We can see in the base case of definition that $\texttt{S}{=}\emptyset$, and in the recursive case that all values stored after `root` must be no less than `root`'s value.

The predicate below is used to specify node-balanced trees with binary search property:

$$\begin{aligned}\texttt{nbt}\langle \texttt{S}\rangle \equiv\ & (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\emptyset) \ \vee \\ & (\texttt{root}{::}\texttt{Node2}\langle \texttt{v}, \texttt{p}, \texttt{q}\rangle * \texttt{p}{::}\texttt{nbt}\langle \texttt{S}_\texttt{p}\rangle * \texttt{q}{::}\texttt{nbt}\langle \texttt{S}_\texttt{q}\rangle \wedge \texttt{S}{=}\{\texttt{s}\} \sqcup \texttt{S}_\texttt{p} \sqcup \texttt{S}_\texttt{q} \wedge \\ & (\forall \texttt{x}{\in}\texttt{S}_\texttt{p}{\cdot}\texttt{x}{\leq}\texttt{s}) \wedge (\forall \texttt{x}{\in}\texttt{S}_\texttt{q}{\cdot}\texttt{s}{\leq}\texttt{x}) \wedge -1{\leq}|\texttt{S}_\texttt{p}|{-}|\texttt{S}_\texttt{q}|{\leq}1)\end{aligned}$$

where `S` captures the content of the tree. We require the difference in node numbers of the left and right sub-trees be within one, as the node-balanced property indicates.

Now let us return to the program itself. Taking a sorted doubly-linked list (`head`) as input, `sdl2nbt` will convert it into a node-balanced tree together with binary search properties, as indicated in lines 2 and 3. Its algorithm proceeds as follows: first it finds the "centre" node in the list (`root`), where the difference of numbers of

```
 0 class Node2 { int val; Node2 prev;
                 Node2 next; }
 1 Node2 sdl2nbt(Node2 head, Node2 tail)
 2  requires head::sdlB⟨p, q, S⟩
 3  ensures  res::nbt⟨S_res⟩
 4 {
 5  Node2 root = head;
 6  Node2 end = head;
 7  while(end != tail) {
 8    end = end.next;
 9    if (end != tail) {
10      end = end.next;
11      root = root.next;
12    }
13  } where head::sdlB⟨p,q,S⟩ ∗⟷ head::sdlB⟨p_h,q_h,S_h⟩
        ∗ root::sdlB⟨p_r, q_r, S_r⟩ ∗ end::sdlB⟨p_e, q_e, S_e⟩
14  if (head == root)
15    root.prev = null;
16  else
17    root.prev = sdl2nbt(head, root);
18  Node2 tmp = root.next;
19  if (tmp == tail)
20    root.next = null;
21  else {
22    tmp.prev = null;
23    root.next = sdl2nbt(tmp, tail);
24  }
25  return root;
26 }
```

Figure 4.2: Algorithm to convert a sorted doubly-linked list to a node-balanced tree.

its left and right nodes is at most one, as Figure 4.3 (a) indicates (lines 5-13). Then it applies the algorithm recursively on both list segments on the centre's left and right hand sides, and regards the centre node as the tree's root, whose left and right children are the resulted subtrees' roots from the recursive calls, as in Figure 4.3 (b) and (c) (lines 14-25). As the data structure of doubly-linked list and binary tree are homomorphic (line 0), we reuse the nodes in the input list instead of creating a new tree, making this algorithm in-place. The parameter `head` in line 1 denotes the first node of the input list, and `tail` is where the list's last node's `next` field points to. When using this method `tail` should be set as `null` initially.

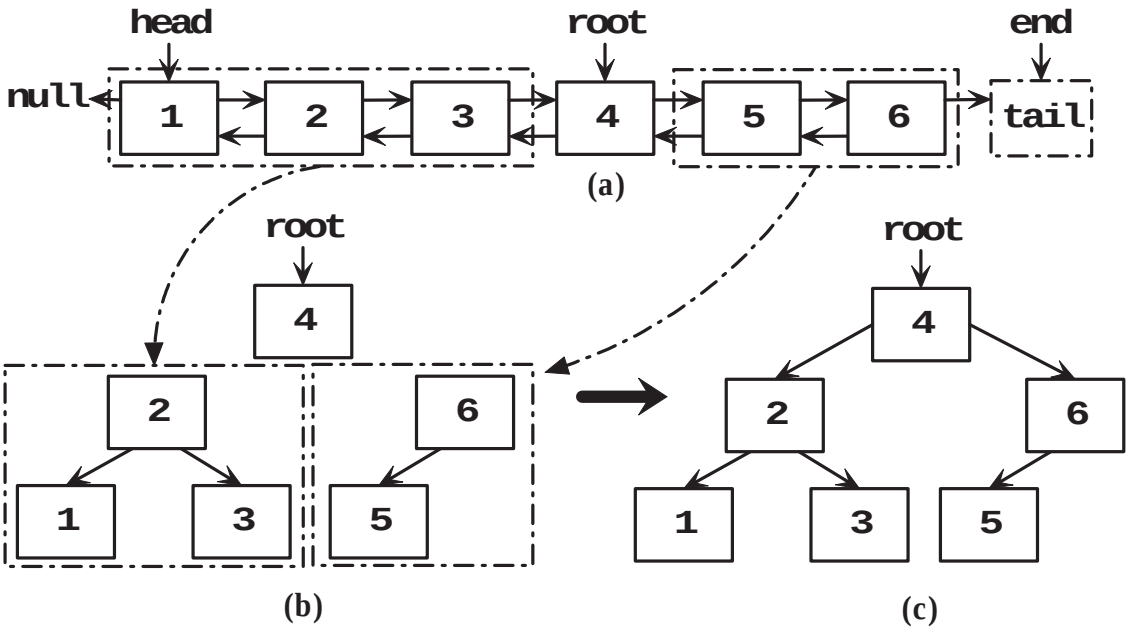

Figure 4.3: Transferring from a sorted doubly-linked list to a node-balanced BST.

Our framework allows the user to verify and/or refine a number of properties about this code. Firstly, the transformation of shapes from initial to final states (namely, from a doubly-linked list to a binary tree) must be captured. Secondly, some structural numerical information should be inferred, so as to prove the node counts before and after the method invocation are the same and the node-balanced property of the tree, etc. Meanwhile, we also want to derive relational numerical information as lists' sortedness and trees' binary search property, and finally multiset information

like the symbolic content of the list's and the tree's (in order to prove the values stored in the list and the resulted tree are the same). Finally, some obligation for memory safety should be found in the precondition, to ensure the input list is non-empty (otherwise the dereference in line 15/17 will fail). To deal with all these properties, we expect the user to provide shape information for the method's (and loop's) specifications as in Figure 4.2. Based on that, we compute the remaining constraints, viz. the missing parts of pure specifications.

As for the example, as the user has provided the pre- and post-shapes for method `sdl2nbt`, our verification proceeds in two steps: generating the constraint abstraction, and solving it. The first step is mainly a forward analysis over the program to find its postcondition, so as to generate the constraint abstraction. Before this step, we assume that the while loop in lines 7-13 is already verified with its specification refined using the same approach. Therefore we take the while loop's postcondition as

$$\begin{aligned}&\mathtt{head}::\mathtt{sdlB}\langle \mathtt{null}, \mathtt{root}, \mathtt{S_h}\rangle * \mathtt{root}::\mathtt{sdlB}\langle \mathtt{p}, \mathtt{tail}, \mathtt{S_r}\rangle \wedge \\ &\quad \mathtt{end}{=}\mathtt{tail} \wedge \mathtt{S}{=}\mathtt{S_h}{\sqcup}\mathtt{S_r} \wedge (\forall \mathtt{x}{\in}\mathtt{S_h}, \mathtt{y}{\in}\mathtt{S_r}{\cdot}\mathtt{x}{\leq}\mathtt{y}) \wedge \underline{0{\leq}|\mathtt{S_r}|{-}|\mathtt{S_h}|{\leq}1}\end{aligned}$$

which indicates that the original list segment starting from `head` is cut into two pieces with a cutpoint `root`, where both are still sorted and the content is also preserved. Meanwhile, the essential constraint (the underlined part, saying the list beginning with `head` is at most one node shorter than that with `root`) to ensure the node-balanced property is derived as well.

When the forward analysis finishes, it generates the following constraint abstraction

as the postcondition of the method:

$$
\begin{array}{l}
\mathtt{Q}(\mathtt{head},\mathtt{p},\mathtt{q},\mathtt{S},\mathtt{res},\mathtt{S_{res}}) ::= \\
\quad \mathtt{root}::\mathtt{Node2}\langle \mathtt{v},\mathtt{null},\mathtt{null}\rangle \wedge \mathtt{head}{=}\mathtt{root}{=}\mathtt{res} \wedge \mathtt{tmp}{=}\mathtt{q}{=}\mathtt{tail} \wedge \mathtt{p}{=}\mathtt{null} \wedge \\
\qquad \mathtt{S}{=}\{\mathtt{v}\} \vee \\
\quad \mathtt{head}::\mathtt{Node2}\langle \mathtt{s},\mathtt{null},\mathtt{root}\rangle * \mathtt{root}::\mathtt{Node2}\langle \mathtt{v},\mathtt{res_h},\mathtt{null}\rangle \wedge \mathtt{res}{=}\mathtt{root} \wedge \\
\qquad \mathtt{tmp}{=}\mathtt{q}{=}\mathtt{tail} \wedge \mathtt{p}{=}\mathtt{null} \wedge \mathtt{S}{=}\{\mathtt{s},\mathtt{v}\} \wedge \mathtt{s}{\leq}\mathtt{v} \vee \\
\quad \mathtt{res_h}::\mathtt{nbt}\langle \mathtt{S^h_{res}}\rangle * \mathtt{res_r}::\mathtt{nbt}\langle \mathtt{S^r_{res}}\rangle * \mathtt{root}::\mathtt{Node2}\langle \mathtt{v},\mathtt{res_h},\mathtt{res_r}\rangle \wedge \\
\qquad \mathtt{P}(\mathtt{head},\mathtt{p},\mathtt{root},\mathtt{S_h},\mathtt{res_h},\mathtt{S^h_{res}}) \wedge \mathtt{P}(\mathtt{tmp},\mathtt{null},\mathtt{tail},\mathtt{S_r},\mathtt{res_r},\mathtt{S^r_{res}}) \wedge \\
\qquad \mathtt{head}{\neq}\mathtt{root} \wedge \mathtt{root}{=}\mathtt{res} \wedge \mathtt{tmp}{\neq}\mathtt{tail} \wedge \mathtt{q}{=}\mathtt{tail} \wedge \\
\qquad \mathtt{S}{=}\mathtt{S_h}{\sqcup}\{\mathtt{v}\}{\sqcup}\mathtt{S_r} \wedge (\forall \mathtt{x}{\in}\mathtt{S_h},\mathtt{y}{\in}\mathtt{S_r}{\cdot}\mathtt{x}{\leq}\mathtt{v}{\leq}\mathtt{y}) \wedge 0{\leq}|\mathtt{S_r}|{-}|\mathtt{S_h}|{\leq}1
\end{array}
$$

where P stands for corresponding pure constraint abstraction as in the previous example. The first two disjunctive branches are base cases of the method's invocation, and the last denotes the effect of recursive calls combined into the postcondition. The first case represents the scenario where there is only one node in the original list (with `res` as the method's return value). The second is for the case of two nodes, one referenced by `head`, pointing to the other one, `root`. In this case the value of `head` is no more than that of `root`. The third case is defined recursively with the constraint abstraction itself, meaning that the post-state concerns the `root` node and the post-states of two recursive calls over `head` and `tmp`, respectively. Note that $\mathtt{S_{res}}$ does not appear in Q's definition. Since it stands for pure properties in user-provided post-shape, it will be involved when we abstract Q against that post-shape in the next step.

The second step solves the constraint abstraction Q by finding its closed-form approximation. Instead of performing a fixed-point iteration directly on Q over the combined domain, we first derive a pure constraint abstraction P (with the help of Sleek) from Q and the user-provided heap part of postcondition. Then we are able to use existing conventional solvers (Nipkow et al., 2002; Popeea and Chin, 2006) to compute the pure fixed-point. For the `sdl2nbt` method, we generate the pure

constraint abstraction P based on the following entailment relation:

$$\mathtt{Q}(\mathtt{head},\mathtt{p},\mathtt{q},\mathtt{S},\mathtt{res},\mathtt{S_{res}}) \vdash \mathtt{res}::\mathtt{nbt}\langle \mathtt{S_{res}}\rangle \wedge \mathtt{P}(\mathtt{head},\mathtt{p},\mathtt{q},\mathtt{S},\mathtt{res},\mathtt{S_{res}})$$

which produces the following pure constraint abstraction P:

$$\begin{array}{l}
\mathtt{P}(\mathtt{head},\mathtt{p},\mathtt{q},\mathtt{S},\mathtt{res},\mathtt{S_{res}}) ::= \\
\quad \mathtt{head}{=}\mathtt{root}{=}\mathtt{res} \wedge \mathtt{tmp}{=}\mathtt{q}{=}\mathtt{tail} \wedge \mathtt{p}{=}\mathtt{null} \wedge \mathtt{S}{=}\mathtt{S_{res}}{=}\{\mathtt{v}\} \vee \\
\quad \mathtt{head}{\neq}\mathtt{root} \wedge \mathtt{res}{=}\mathtt{root} \wedge \mathtt{tmp}{=}\mathtt{q}{=}\mathtt{tail} \wedge \mathtt{p}{=}\mathtt{null} \wedge \\
\qquad \mathtt{S}{=}\mathtt{S_{res}}{=}\{\mathtt{s},\mathtt{v}\} \wedge \mathtt{s}{\leq}\mathtt{v} \vee \\
\quad \mathtt{P}(\mathtt{head},\mathtt{p},\mathtt{root},\mathtt{S_h},\mathtt{res_h},\mathtt{S^h_{res}}) \wedge \mathtt{P}(\mathtt{tmp},\mathtt{null},\mathtt{tail},\mathtt{S_r},\mathtt{res_r},\mathtt{S^r_{res}}) \wedge \\
\qquad \mathtt{head}{\neq}\mathtt{root} \wedge \mathtt{root}{=}\mathtt{res} \wedge \mathtt{tmp}{\neq}\mathtt{tail} \wedge \mathtt{q}{=}\mathtt{tail} \wedge \mathtt{S}{=}\mathtt{S_h}{\sqcup}\{\mathtt{v}\}{\sqcup}\mathtt{S_r} \wedge \\
\qquad \mathtt{S_{res}}{=}\mathtt{S^h_{res}}{\sqcup}\{\mathtt{v}\}{\sqcup}\mathtt{S^r_{res}} \wedge (\forall \mathtt{x}{\in}\mathtt{S_h},\mathtt{y}{\in}\mathtt{S_r}{\cdot}\mathtt{x}{\leq}\mathtt{v}{\leq}\mathtt{y}) \wedge 0{\leq}|\mathtt{S_r}|{-}|\mathtt{S_h}|{\leq}1
\end{array}$$

Note that the heap information is already eliminated from P; instead the constraints over $\mathtt{S_{res}}$ are included during the entailment checking procedure. This allows us to solve P to refine the user-provided shape-only specification.

After solving P, we achieve the following constraint:

$$\mathtt{p}{=}\mathtt{null} \wedge \mathtt{q}{=}\mathtt{tail} \wedge \mathtt{S}{=}\mathtt{S_{res}} \wedge |\mathtt{S}|{\geq}1$$

with which we can refine the method's specifications as

$$\begin{array}{ll}
\mathtt{requires} & \mathtt{head}::\mathtt{sdlB}\langle \mathtt{p},\mathtt{q},\mathtt{S}\rangle \wedge \mathtt{p}{=}\mathtt{null} \wedge \mathtt{q}{=}\mathtt{tail} \wedge |\mathtt{S}|{\geq}1 \\
\mathtt{ensures} & \mathtt{res}::\mathtt{nbt}\langle \mathtt{S_{res}}\rangle \wedge \mathtt{S}{=}\mathtt{S_{res}}
\end{array}$$

which proposes more requirements in the precondition, as the `head`'s `prev` field should be `null`, and the whole list's last node's `next` field must point to `tail`. Meanwhile, there should be at least one node in the list for the sake of memory safety. With those obligations, the method guarantees that the result is a node-balanced tree with binary search property, whose content is the same as the input list.

4.3 The Verification

In this section, we formulate our verification algorithm for methods with partial specifications given, together with the pure abduction mechanism and forward analysis rules it uses.

4.3.1 Refining Partial Specifications

Algorithm CA_Gen_Solve($\mathcal{T}, mn, e, \Phi_{pr}, \Phi_{po}, \boldsymbol{u}, \boldsymbol{v}$)

1 Δ := Symb_Exec($\mathcal{T}, mn, e, \Phi_{pr}$)

2 **if** Δ = fail **then return** fail **end if**

3 Normalise Δ to DNF, and denote as $\bigvee_{i=1}^{m} \Delta_i$

4 $\boldsymbol{w}$:=$\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{v'}\} \cup$ pureV($\{\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{v'}\}$, $\Phi_{pr} \vee \Phi_{po}$)

5 $\Delta_{\mathtt{P}}$:= Pure_CA_Gen(Φ_{po}, $\mathtt{Q}(\boldsymbol{w})$::=$\bigvee_{i=1}^{m} \Delta_i$)

6 **if** $\Delta_{\mathtt{P}}$ = fail **then return** fail **end if**

7 π := Pure_CA_Solve($\mathtt{P}(\boldsymbol{w})$::=$\Delta_{\mathtt{P}}$)

8 $R := t\ mn\ ((\boldsymbol{t}\ \boldsymbol{u}); (\boldsymbol{t}\ \boldsymbol{v}))$ *requires* ex_quan(Φ_{pr}, π) *ensures* ex_quan(Φ_{po}, π)

9 **if** HipVerify($\mathcal{T}, mn, R$) **then return** $\mathcal{T} \cup \{R\} \setminus \{\, t\ mn\ ((\boldsymbol{t}\ \boldsymbol{u}); (\boldsymbol{t}\ \boldsymbol{v}))\ \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po}\,\}$

10 **else return** fail **end if**

end Algorithm

Figure 4.4: Refining method specifications.

The algorithm for refinement (CA_Gen_Solve) is given in Figure 4.4. As illustrated

in Section 4.2, the verification proceeds in two steps for a method with shape information given in specification, namely (1) forward analysis (at lines 1-2) and (2) pure constraint abstraction generation and solving (at lines 3-10).

Lines 1-2 analyse the method body starting from the given pre-shape to compute the post-state in constraint abstraction form. Along the analysis, missing pure requirements are derived (via our abduction mechanism) to strengthen the precondition.

The forward analysis (line 3 in Figure 4.5 invoked by line 1 in Figure 4.4) is conducted using a set of symbolic execution rules to be explained in Section 4.3.3. If the symbolic execution of the method body succeeds (suggesting that the pre-shape is sufficiently strong), the verification moves on to the second step (lines 3-10). However, if the symbolic execution fails at some point, where the current symbolic state cannot meet the requirement of the next instruction, it can be due to the lack of pure (i.e. numerical/multiset) constraints in the precondition. To deal with this, we enhance the symbolic execution with *pure abduction mechanism* (whose details are given later). For example, if we have $\texttt{x::ll}\langle\texttt{n}\rangle$ as the current state and we require $\texttt{x::Node}\langle\_, \texttt{p}\rangle$ to update the value of p, then it will fail as $\texttt{x::ll}\langle\texttt{n}\rangle$ does not necessarily guarantee $\texttt{x::Node}\langle\_, \texttt{p}\rangle$. In this case we conduct the pure abduction as

$$\texttt{x::ll}\langle\texttt{n}\rangle \wedge [\texttt{n}{\geq}\texttt{1}] \rhd \texttt{x::Node}\langle\_, \texttt{p}\rangle * \texttt{true}$$

to compute the missing pure information (in the squared bracket) such that the left hand side (including the newly gained pure part) entails the right hand side.

The Symb_Exec in Figure 4.5 is our symbolic execution algorithm. The variable *errLbls* (initialised at line 1) is to record the program locations in which previous pure abductions occurred. Whenever the symbolic execution fails, it returns a state Δ that contains the pure abduction result and the location l where failure was detected, as shown in line 3. If the current abduction location l is not recorded in *errLbls*, it

Algorithm $\mathsf{Symb\_Exec}(\mathcal{T}, mn, e, \Phi_{pr})$

1 $errLbls := \emptyset$

2 **do**

3 $(\Delta, l) := [\![e]\!]_{\mathcal{T}}(\Phi_{pr}, 0)$

4 **if** $l{>}0 \wedge l{\notin} errLbls$ **then**

5 $\mathcal{T} := \mathcal{T} \setminus \{t\ mn\ ((t_i\ u_i)_{i=1}^{m}; (t_i\ v_i)_{i=1}^{n})\ requires\ \Phi_{pr}\ ensures\ \Phi_{po}\}$

6 $\Phi_{pr} := \mathsf{ex\_quan}(\Phi_{pr}, \Delta)$

7 $\mathcal{T} := \mathcal{T} \cup \{t\ mn\ ((t_i\ u_i)_{i=1}^{m}; (t_i\ v_i)_{i=1}^{n})\ requires\ \Phi_{pr}\ ensures\ \Phi_{po}\}$

8 $errLbls := errLbls \cup \{l\}$

9 **else if** $l{>}0 \wedge l{\in} errLbls$ **then return** fail

10 **end if**

11 **while** $l > 0$

12 **return** Δ

end Algorithm

Figure 4.5: Symbolic execution.

indicates that this is a new failure. The abduction result is added to the precondition of the current method to obtain a stronger Φ_{pr} (which also replaces the previous one in $\mathcal{T}$), before the algorithm enters again the symbolic execution loop with variable *errLbls* updated to add in the new failure location l.[2] This loop is repeated until symbolic execution succeeds with no memory error, or a previous failure point was re-encountered. The latter indicates either a program bug or a specification error. For example, for a method `void foo` (...) {`node w` = `new node(0, null); goo(w);` ...} invoking a method `goo(x)` whose precondition is `x::ll`$\langle$`n`$\rangle\wedge$`n`$\geq$`2`, our verification will perform an abduction to get `n`$\geq$`2` since it is not implied by the current state. However, as `n` is for the shape of local variable `w`, it will be quantified away when `n`$\geq$`2` is propagated back, ending up with `true` being added to `foo`'s precondition. In the next round of symbolic execution, our verification will have the same abduction at the same point. Such case is reported as fail. In this way, the symbolic execution continues until it reports an error or reaches the end of the method body (exiting line 11).

Back to the main algorithm CA_Gen_Solve, the verification next builds a heap-based constraint abstraction mechanism, named `Q`($\boldsymbol{w}$), for the post-state in lines 3-5. This constraint abstraction is possibly recursive. We then make use of another algorithm in Figure 4.6, named Pure_CA_Gen, to extract a pure constraint abstraction `P`($\boldsymbol{w}$) without any heap property. This algorithm tries to derive a branch $\texttt{P}_i$ for each branch Δ_i of `Q`. For every Δ_i it proceeds in two steps. In the first step (lines 2-4), it replaces the recursive occurrence of `Q` in Δ_i with $\sigma * \texttt{P}(\boldsymbol{w})$. In the second step (lines 5-7) it tries to derive $\texttt{P}_i$ via the entailment. If the entailment fails, then pure abduction is used to discover any missing pure constraint σ'_i for $\rho\Delta_i$ to allow the entailment to succeed. In this case, σ'_i is incorporated into σ_i (and eventually $\texttt{P}_i$).

[2] ex_quan(Φ_{pr}, *XPure*(Δ)) is to combine Φ_{pr} with a pure approximation of Δ where *XPure* is a strengthened version of that in Nguyen et al. (2007), as it also takes pure parts in Δ and keeps them in the resulted pure constraints.

Once this is done, we use some existing fixed-point analysis (Nipkow et al., 2002; Popeea and Chin, 2006) to derive non-recursive constraint π, as a simplification of $\mathtt{P}(\boldsymbol{w})$. This result is then incorporated into the pre/post specifications in line 8, before we perform a post-verification in line 9 using the HIP verifier (Chin et al., 2010), to ensure the strengthened precondition is strong enough for memory safety.

Algorithm Pure_CA_Gen($\sigma, \mathtt{Q}(\boldsymbol{w}) ::= \bigvee_{i=1}^{m} \Delta_i$)

1 **for** $i = 1$ **to** m

2 Denote all appearances of $\mathtt{Q}(\boldsymbol{w})$ in Δ_i as $\mathtt{Q}_j(\boldsymbol{w_j}), j = 1, ..., p$

3 Denote substitutions $\rho_j = [([\boldsymbol{w_j}/\boldsymbol{w}]\sigma * \mathtt{P}(\boldsymbol{w_j}))/\mathtt{Q}_j(\boldsymbol{w_j})]$

4 Let substitution $\rho := \rho_1 \circ \rho_2 \circ ... \circ \rho_p$ as applying all substitutions defined above in sequence

5 **if** $(\rho\Delta_i \vdash \sigma * \sigma_i$ **or** $\rho\Delta_i \wedge [\sigma_i'] \rhd \sigma * \sigma_i)$ **and** $ispure(\sigma_i)$

6 **then** $\mathtt{P}_i := \sigma_i$

7 **else return** fail **end if**

8 **end for**

9 **return** $\bigvee_{i=1}^{m} \mathtt{P}_i$

end Algorithm

Figure 4.6: Pure constraint abstraction generation algorithm.

Two auxiliary functions used in the algorithm are described here. The function $\mathsf{pureV}(V, \Delta)$ retrieves from Δ the shapes referred to by all pointer variables from V, and returns the set of logical variables used to record numerical (size and bag) properties in these shapes, for example, $\mathsf{pureV}(\{\mathtt{x}\}, \mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle)$ returns $\{\mathtt{n}\}$. This function is used in the algorithm to ensure that all free variables in Φ_{pr} and Φ_{po} are added into the parameter list of the constraint abstraction $\mathtt{Q}$. The function $\mathsf{ex\_quan}(\Delta, \pi)$

is to strengthen the state Δ with the abduction result π:

$$\mathsf{ex\_quan}(\Delta, \pi) =_{df} \Delta \wedge \exists(\mathsf{fv}(\pi) \setminus \mathsf{fv}(\Delta)) \cdot \pi$$

It is used to incorporate the discovered missing pure constraints into the original specification. For example, $\mathsf{ex\_quan}(\mathtt{x{::}ll\langle n\rangle}, \mathtt{0{<}m} \wedge \mathtt{m{\leq}n})$ returns $\mathtt{x{::}ll\langle n\rangle} \wedge \mathtt{0{<}n}$.

4.3.2 Pure Abduction Mechanism

We assume that the user has supplied necessary shape information in the specifications for primary methods. When an entailment fails (during symbolic execution or pure constraint abstraction derivation), we use our pure abduction mechanism (Figure 4.7) to discover missing pure constraints. Note that we focus on pure abduction in this chapter (as it is sufficient and efficient for our approach), though it might be possible to adapt the shape abduction technique (Calcagno et al., 2009) (to those with strong invariants) in case that shape information is missing from the given precondition, which will be introduced in Chapter 6. For example, if we have $\mathtt{x{::}ll\langle n\rangle}$ as the current state and we require $\exists \mathtt{v, p} \cdot \mathtt{x{::}Node\langle v, p\rangle}$ to update the value of $\mathtt{p}$, then it will fail as $\mathtt{x{::}ll\langle n\rangle}$ does not necessarily guarantee $\exists \mathtt{v, p} \cdot \mathtt{x{::}Node\langle v, p\rangle}$. In this case we conduct the pure abduction as

$$\mathtt{x{::}ll\langle n\rangle} * [\mathtt{n{\geq}1}] \rhd \exists \mathtt{v, p} \cdot \mathtt{x{::}Node\langle v, p\rangle} * \mathtt{true}$$

to compute the missing pure information (in the squared bracket) such that the left hand side (including the newly gained pure part) entails the right hand side. Later this information will be used to strengthen the program's precondition.

Our pure abduction deals with three different cases. The first rule applies when the left hand side (σ) does not entail the right hand side (σ_1) but the right hand side entails the left hand side with some pure formula (σ') as the frame; for instance, in $\mathtt{x{::}ll\langle n\rangle} \not\vdash \mathtt{x{::}Node\langle \_, null\rangle}$, the right hand side can entail the left hand

$$\frac{\sigma \not\vdash \sigma_1 * \texttt{true} \quad \sigma_1 \vdash \sigma * \sigma' \quad \mathit{ispure}(\sigma') \quad \sigma \wedge \sigma' \vdash \sigma_1 * \sigma_2}{\sigma \wedge [\sigma'] \rhd \sigma_1 * \sigma_2}$$

$$\frac{\begin{array}{ll} \sigma \not\vdash \sigma_1 * \texttt{true} & \sigma_1 \not\vdash \sigma * \texttt{true} \\ \sigma_0 \in \mathsf{unroll}(\sigma) & \mathsf{data\_no}(\sigma_0) \leq \mathsf{data\_no}(\sigma_1) \\ \sigma_0 \vdash \sigma_1 * \sigma' \ \text{ or } & \sigma_0 \wedge [\sigma_0'] \rhd \sigma_1 * \sigma' \\ \mathit{ispure}(\sigma') & \sigma \wedge \sigma' \vdash \sigma_1 * \sigma_2 \end{array}}{\sigma \wedge [\sigma'] \rhd \sigma_1 * \sigma_2}$$

$$\frac{\begin{array}{c} \sigma \not\vdash \sigma_1 * \texttt{true} \qquad \sigma_1 \not\vdash \sigma * \texttt{true} \\ \sigma_1 \wedge [\sigma_1'] \rhd \sigma * \sigma' \quad \mathit{ispure}(\sigma') \\ \sigma \wedge \sigma' \vdash \sigma_1 * \sigma_2 \end{array}}{\sigma \wedge [\sigma'] \rhd \sigma_1 * \sigma_2}$$

Figure 4.7: Pure abduction rules.

side with pure frame $\texttt{n=1}$. The abduction then checks to ensure $\texttt{x::ll}\langle\texttt{n}\rangle \wedge \texttt{n=1} \vdash \texttt{x::Node}\langle\_, \texttt{null}\rangle * \sigma_2$ for some σ_2, and returns the result $\texttt{n=1}$. Note the check $\mathit{ispure}(\sigma')$ ensures that σ' contains no heap information.

In the second rule, neither side entails the other (first row), say $\sigma = \texttt{x::sllB}\langle\texttt{S}\rangle$ and $\sigma_1 = \exists \texttt{p}, \texttt{u}, \texttt{v} \cdot \texttt{x::Node}\langle\texttt{u}, \texttt{p}\rangle * \texttt{p::Node}\langle\texttt{v}, \texttt{null}\rangle$. As the shape predicates in the antecedent are formed by disjunctions according to their definitions (like the `sllB`), certain branches of σ may entail σ_1. As the rule suggests, to accomplish abduction $\sigma \wedge [\sigma'] \rhd \sigma_1 * \sigma_2$, we first unfold σ (second row) and try entailment or further abduction with the results (σ_0) against σ_1 (third row). If it succeeds with a pure frame σ', then we confirm the abduction by checking $\sigma \wedge \sigma' \vdash \sigma_1 * \sigma_2$ (fourth row). For the example above, the abduction returns $\texttt{|S|=2}$ (σ') and discovers the nontrivial frame $\texttt{S=}\{\texttt{u}, \texttt{v}\} \wedge \texttt{u}\leq\texttt{v}$ (σ_2). Note the function $\mathsf{data\_no}$ returns the number of object nodes in a state, for instance it returns one for $\texttt{x::Node}\langle\texttt{v}, \texttt{p}\rangle * \texttt{p::ll}\langle\texttt{m}\rangle$. (This syntac-

tic check is important for the termination of the abduction.) The unroll unfolds all shape predicates once in σ, normalises the result to a disjunctive form ($\bigvee_{i=1}^{u} \sigma^i$), and returns the result as a set of formulae ($\{\sigma^1, ..., \sigma^u\}$). An instance is that it expands $\mathtt{x}{::}\mathtt{Node}\langle \mathtt{v}, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{ll}\langle \mathtt{m}\rangle$ to be $\{\mathtt{x}{::}\mathtt{Node}\langle \mathtt{v}, \mathtt{p}\rangle \wedge \mathtt{p}{=}\mathtt{null} \wedge \mathtt{m}{=}\mathtt{0}, \exists \mathtt{u}, \mathtt{q}, \mathtt{k} \cdot \mathtt{x}{::}\mathtt{Node}\langle \mathtt{v}, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{Node}\langle \mathtt{u}, \mathtt{q}\rangle * \mathtt{q}{::}\mathtt{ll}\langle \mathtt{k}\rangle \wedge \mathtt{m}{=}\mathtt{k}{+}\mathtt{1}\}$.

In the third rule, neither side entails the other, and the second rule does not apply, This happens frequently during the abstraction stage in the verification when we need to fold up a "concrete" state of nodes against an abstracted shape predicate, say, $\sigma = \exists \mathtt{p}, \mathtt{u}, \mathtt{v} \cdot \mathtt{x}{::}\mathtt{Node}\langle \mathtt{u}, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{Node}\langle \mathtt{v}, \mathtt{null}\rangle$, $\sigma_1 = \exists \mathtt{S} \cdot \mathtt{x}{::}\mathtt{sllB}\langle \mathtt{S}\rangle$. In this case the antecedent cannot be unfolded as they are object nodes. As the rule suggests, it reverses two sides of the entailment and applying the second rule to uncover the pure constraints σ_1' and σ' (second row). It checks that adding σ' to the left hand side (σ) does entail the right hand side (σ_1) (third row) before it returns σ'. For the example above, the abduction returns $\mathtt{u}{\leq}\mathtt{v}$ which is essential for the two nodes to form a sorted list (right hand side).

4.3.3 Symbolic Execution Rules

This section defines the symbolic execution rules used in the first step of the constraint abstraction generation. If the program contains recursive calls to itself, the postcondition will be in a recursive (open) form.

The type of our symbolic execution is defined as

$$[\![e]\!] =_{df} \mathsf{AllSpec} \rightarrow (\mathcal{P}_{\mathsf{SH}} \times \mathsf{Int}) \rightarrow (\mathcal{P}_{\mathsf{SH}} \times \mathsf{Int})$$

where AllSpec contains all the specifications of all methods (extracted from the program *Prog*). The integer (label) in both input and output is used to record a

program location where abduction is needed. If the integer remains zero after the symbolic execution of e, then the output state denotes the post-state of e. However, a positive number indicates that an abduction must have occurred and the resulting state (the abduction result) will be propagated back to the the method's precondition by our verification, so that the next round of symbolic execution should succeed in the same location.

The foundation of the symbolic execution is the basic transition functions from a conjunctive abstract state to a conjunctive or disjunctive abstract state below:

$$\begin{array}{lll} \mathsf{unfold}(x) & =_{df} \mathsf{SH} \rightarrow \mathcal{P}_{\mathsf{SH}[x]} & \text{Unfolding} \\ \mathsf{exec}(d[x]) & =_{df} \mathsf{AllSpec} \rightarrow (\mathsf{SH}[x] \times \mathsf{Int}) \rightarrow (\mathsf{SH} \times \mathsf{Int}) & \text{Heap-sensitive execution} \\ \mathsf{exec}(d) & =_{df} \mathsf{AllSpec} \rightarrow (\mathsf{SH} \times \mathsf{Int}) \rightarrow (\mathsf{SH} \times \mathsf{Int}) & \text{Heap-insensitive execution} \end{array}$$

where $\mathsf{SH}[x]$ denotes the set of conjunctive abstract states in which each element has x exposed as the head of an object node ($x{::}c\langle \boldsymbol{v} \rangle$), and $\mathcal{P}_{\mathsf{SH}[x]}$ contains all the (disjunctive) abstract states, each of which is composed by such conjunctive states. Here $\mathsf{unfold}(x)$ unfolds the symbolic heap so that the cell referred to by x is exposed for access by heap sensitive commands $d[x]$ via the second transition function $\mathsf{exec}(d[x])$. The third function defined for other (heap insensitive) commands d does not require such exposure of x.

For the unfolding operation $\mathsf{unfold}(x)$, there are two possible scenarios. If `x` refers to an object node in the current state σ, no unfolding is required and the exec operation can proceed directly. However, if `x` refers to a (user-defined) shape predicate, then $\mathsf{unfold}(x)$ will unfold the current state σ according to the definition of the predicate in order to expose the object node referred to by `x`:

$$\frac{\mathit{isobj}(c) \quad \sigma \vdash x{::}c\langle \boldsymbol{v} \rangle * \sigma'}{\mathsf{unfold}(x)\sigma \rightsquigarrow \sigma} \qquad \frac{\mathit{isspred}(c) \quad \sigma \vdash x{::}c\langle \boldsymbol{u} \rangle * \sigma' \quad \texttt{root}{::}c\langle \boldsymbol{v} \rangle \equiv \Phi}{\mathsf{unfold}(x)\sigma \rightsquigarrow \sigma' * [x/\texttt{root}, \boldsymbol{u}/\boldsymbol{v}]\Phi}$$

The test $\mathit{isobj}(c)$ returns `true` only if c is an object node and $\mathit{isspred}(c)$ returns `true`

only if c is a shape predicate.

The symbolic execution of heap-sensitive commands $\texttt{d[x]}$ (i.e. $\texttt{x.f}$, $\texttt{x.f := w}$, or $\texttt{free(x)}$) assumes that the unfolding $\mathsf{unfold}(x)$ has been done prior to the execution. The first three rules below are for normal symbolic execution where the current state is sufficiently strong for safe execution. The last two rules handle the cases where the symbolic execution fails and abductive reasoning can be used to discover missing pure information.

$$\frac{\mathit{isobj}(c) \qquad \sigma \vdash x{::}c\langle v_1, .., v_n\rangle * \sigma'}{\mathsf{exec}(x.f_i)(\mathcal{T})(\sigma, 0) \rightsquigarrow (\sigma' * x{::}c\langle v_1, .., v_n\rangle \wedge \texttt{res}{=}v_i, 0)}$$

$$\frac{\mathit{isobj}(c) \qquad \sigma \vdash x{::}c\langle v_1, .., v_n\rangle * \sigma'}{\mathsf{exec}(x.f_i := w)(\mathcal{T})(\sigma, 0) \rightsquigarrow (\sigma' * x{::}c\langle v_1, .., v_{i-1}, w, v_{i+1}, .., v_n\rangle, 0)}$$

$$\frac{\mathit{isobj}(c) \qquad \sigma \vdash x{::}c\langle \boldsymbol{u}\rangle * \sigma'}{\mathsf{exec}(\texttt{free}(x))(\mathcal{T})(\sigma, 0) \rightsquigarrow (\sigma', 0)}$$

$$\frac{\mathit{isobj}(c) \quad \sigma \nvdash x{::}c\langle \boldsymbol{u}\rangle{*}\texttt{true} \quad \sigma{*}[\sigma'] \rhd x{::}c\langle \boldsymbol{u}\rangle{*}\texttt{true}}{\mathsf{exec}(d[x])(\mathcal{T})(\sigma, 0) \rightsquigarrow (\sigma', \mathit{lbl}(d[x]))}$$

$$\frac{\mathit{isobj}(c) \quad \sigma \nvdash x{::}c\langle \boldsymbol{u}\rangle{*}\texttt{true} \quad \sigma{*}[\sigma'] \not\rhd x{::}c\langle \boldsymbol{u}\rangle{*}\texttt{true}}{\mathsf{exec}(d[x])(\mathcal{T})(\sigma, 0) \rightsquigarrow (\texttt{false}, \mathit{lbl}(d[x]))}$$

Note that the second to last rule uses an abductive reasoning (via Sleek) to discover the missing numerical information σ'. Here we use a mapping $\mathit{lbl}(-)$ to map any instruction in the program being analysed to a unique positive integer label (namely the aforementioned program location). The rule changes the second element of the result to $\mathit{lbl}(d[x])$ which will be used by the verification to record the instruction causing an abduction, quits the current execution, propagates the discovered information back to the precondition of the current method, and restarts the symbolic execution with the strengthened precondition. The last rule covers the scenario in which the abduction fails. Then the execution cannot continue and returns $(\texttt{false}, \mathit{lbl}(d[x]))$.

The symbolic execution rules for heap-insensitive commands are as follows:

$$\mathsf{exec}(k)(\mathcal{T})(\sigma,0) \rightsquigarrow (\sigma \wedge \texttt{res}{=}k, 0) \qquad \mathsf{exec}(v)(\mathcal{T})(\sigma,0) \rightsquigarrow (\sigma \wedge \texttt{res}{=}v, 0)$$

$$\frac{\mathit{isobj}(c)}{\mathsf{exec}(\texttt{new}\ c(\boldsymbol{v}))(\mathcal{T})(\sigma,0) \rightsquigarrow (\sigma * \texttt{res}{::}c\langle\boldsymbol{v}\rangle, 0)}$$

$$\frac{\begin{array}{c} t\ mn\ ((t_i\ u_i)_{i=1}^m; (t_i\ v_i)_{i=1}^n)\ \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \in \mathcal{T} \\ \rho = [x_i'/u_i]_{i=1}^m \circ [y_i'/v_i]_{i=1}^n \qquad \sigma \vdash \rho\Phi_{pr} * \sigma' \\ \rho_o = [r_i/v_i]_{i=1}^n \circ [x_i'/u_i']_{i=1}^m \circ [y_i'/v_i']_{i=1}^n \quad \rho_l = [r_i/y_i']_{i=1}^n \quad \mathit{fresh\ logical}\ r_i \end{array}}{\mathsf{exec}(mn(x_1..x_m; y_1..y_n))(\mathcal{T})(\sigma,0) \rightsquigarrow ((\rho_l\,\sigma') * (\rho_o\,\Phi_{po}), 0)}$$

$$\frac{\begin{array}{c} t\ mn\ ((t_i\ u_i)_{i=1}^m; (t_i\ v_i)_{i=1}^n)\ \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \in \mathcal{T} \\ \rho = [x_i'/u_i]_{i=1}^m \circ [y_i'/v_i]_{i=1}^n \qquad \sigma \vdash \rho\Phi_{pr} * \sigma' \\ \rho_o = [r_i/v_i]_{i=1}^n \circ [x_i'/u_i']_{i=1}^m \circ [y_i'/v_i']_{i=1}^n \quad \rho_l = [r_i/y_i']_{i=1}^n \quad \mathit{fresh\ logical}\ r_i \end{array}}{\mathsf{exec}(mn(x_1..x_m; y_1..y_n))(\mathcal{T})(\sigma,0) \rightsquigarrow ((\rho_l\,\sigma') * (\rho_o\,(\Phi_{po} \wedge \mathtt{P}(\boldsymbol{u},\boldsymbol{v}))), 0)}$$

$$\frac{t\ mn...{\in}\mathcal{T} \quad \rho = [x_i'/u_i]_{i=1}^m \circ [y_i'/v_i]_{i=1}^n \quad \sigma \not\vdash \rho\Phi_{pr} * \texttt{true} \quad \sigma \wedge [\sigma'] \rhd \rho\Phi_{pr} * \texttt{true}}{\mathsf{exec}(mn(x_1..x_m; y_1..y_n))(\mathcal{T})(\sigma,0) \rightsquigarrow (\sigma', \mathit{lbl}(mn(...)))}$$

$$\frac{t\ mn...{\in}\mathcal{T} \quad \rho = [x_i'/u_i]_{i=1}^m \circ [y_i'/v_i]_{i=1}^n \quad \sigma \not\vdash \rho\Phi_{pr} * \texttt{true} \quad \sigma \wedge [\sigma'] \not\rhd \rho\Phi_{pr} * \texttt{true}}{\mathsf{exec}(mn(x_1..x_m; y_1..y_n))(\mathcal{T})(\sigma,0) \rightsquigarrow (\texttt{false}, \mathit{lbl}(mn(...)))}$$

Note that the first three rules deal with constant (k), variable (v) and object node creation ($\texttt{new}\ c(\boldsymbol{v})$), respectively, while the remaining rules handle method invocation. The fourth rule is used for the invocation of another method mn which has already been annotated, and the call site meets the precondition of mn, as checked by the entailment $\sigma \vdash \rho\Phi_{pr} * \sigma'$. In this case, the execution succeeds and moves on. The fifth rule is for a recursive call to the current method, similar as above except that a constraint abstraction is in place as postcondition. The last two rules are for the cases where the call site cannot establish the precondition of the callee method and where abductive reasoning is employed. In both cases, the execution discontinues. The sixth rule returns the abduction result σ', which is a pure formula

and will be propagated back by the verification to strengthen the caller method's precondition. The last rule captures the scenario in which the abduction fails. Note that the operator $\circ$ is used to compose two substitutions: the substitution $\rho_2 \circ \rho_1$ works by first applying ρ_1 and then ρ_2.

To keep presentation simple, we assume there are no mutual recursions in the programs to analyse; therefore each method to be analysed should only call itself recursively. This assumption does not lose generality, as we can always transform mutual recursion into single recursion (Rubio-Sánchez et al., 2008) to have only one constraint abstraction $\mathtt{Q}$ in our verification for one method.

The following rule for all commands signifies that when starting from a configuration in which the second element is positive (i.e. a faulty state), the execution will not change the state. This rule is used to skip all remaining instructions when abductive reasoning is used as a new round of symbolic execution with strengthened precondition should be started instead:

$$\frac{l > 0}{\mathsf{exec}(-)(\mathcal{T})(\sigma, l) \rightsquigarrow (\sigma, l)}$$

We can now lift unfold's domain to $\mathcal{P}_{\mathsf{SH}}$ using the following operation $\mathsf{unfold}^{\dagger}$:

$$\mathsf{unfold}^{\dagger}(x) \bigvee \sigma_i =_{df} \bigvee (\mathsf{unfold}(x)\sigma_i)$$

and similarly for exec:

$$\mathsf{exec}^{\dagger}(d)(\mathcal{T})(\bigvee \sigma_i, l) =_{df} \quad (\bigvee \sigma'_i,\, \mathsf{max}\{l_i\}) \text{ where } (\sigma'_i, l_i) = \mathsf{exec}(d)(\mathcal{T})(\sigma_i, l)$$

The symbolic execution rules for program constructors e can now be defined using the lifted transition functions above. Firstly, no change will be made if starting from a faulty state, as the first rule shows. In all other cases, the symbolic execution

transforms one abstract state to another w.r.t. the program instruction:

$$
\begin{array}{lll}
[\![-]\!]_{\mathcal{T}}(\Delta, l) & =_{df} & (\Delta, l), \quad \text{where } l > 0 \\
[\![d[x]]\!]_{\mathcal{T}}(\Delta, 0) & =_{df} & \mathsf{exec}^{\dagger}(d[x])(\mathcal{T})(\mathsf{unfold}^{\dagger}(x)\Delta, 0) \\
[\![d]\!]_{\mathcal{T}}(\Delta, 0) & =_{df} & \mathsf{exec}^{\dagger}(d)(\mathcal{T})(\Delta, 0) \\
[\![e_1; e_2]\!]_{\mathcal{T}}(\Delta, 0) & =_{df} & [\![e_2]\!]_{\mathcal{T}} \circ [\![e_1]\!]_{\mathcal{T}}(\Delta, 0)
\end{array}
$$

$$
[\![v := e]\!]_{\mathcal{T}}(\Delta, 0) \;=_{df}\; [v_1/v', r_1/\mathtt{res}]([\![e]\!]_{\mathcal{T}}(\Delta, 0)) \wedge v'{=}r_1, \; \textit{fresh } v_1, r_1
$$

$$
\frac{(\Delta_1', l_1) = [\![e_1]\!]_{\mathcal{T}}(v{\wedge}\Delta, 0) \qquad (\Delta_2', l_2) = [\![e_2]\!]_{\mathcal{T}}(\neg v{\wedge}\Delta, 0)}{[\![\mathtt{if}\ (v)\ e_1\ \mathtt{else}\ e_2]\!]_{\mathcal{T}}(\Delta, 0) =_{df} (\Delta_1' \vee \Delta_2', \mathsf{max}\{l_1, l_2\})}
$$

4.3.4 Soundness

We have defined the underlying operational semantics of our language in Chapter 3. Its concrete program state consists of stack s and heap h. We have also defined the relation $s, h \models \Delta$ and the transition $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', \nu\rangle$. Before proceeding to the soundness definition, recalling that we have both unprimed variables (for their initial values in abstract states) and primed ones (for their current values), we realise that the concrete program states should always be linked to the primed ones. For this reason we have the following definition:

Definition 4.3.1 (Poststate) *Given an abstract state Δ, Post(Δ) captures the relation between primed variables of Δ. That is,*

$$
\begin{array}{l}
\mathit{Post}(\Delta) =_{df} \rho(\exists V \cdot \Delta), \ \textit{where} \\
V = \{v_1, \ldots, v_n\} \ \textit{denotes all unprimed program variables in } \Delta, \ \textit{and} \\
\rho = [v_1/v_1', \ldots, v_n/v_n'].
\end{array}
$$

For example, for $\Delta = \mathtt{x'{::}Node\langle v', y'\rangle} \wedge \mathtt{v'{=}v} \wedge \mathtt{y'{=}null}$, *we have* $\mathit{Post}(\Delta) = \mathtt{x{::}Node\langle v, y\rangle} \wedge \mathtt{y{=}null}$.

Then we define the soundness of our refinement as follows:

Definition 4.3.2 (Soundness) *For a method definition* $t\ mn\ ((\boldsymbol{t\ u});(\boldsymbol{t\ v}))\ \{e\}$, *if our verification refines its specification as* $t\ mn\ ((\boldsymbol{t\ u});(\boldsymbol{t\ v}))$ *requires* Φ_{pr} *ensures* $\Phi_{po}\ \{e\}$, *then for all* $s,h \models Post(\Phi_{pr})$, *if* $\langle s,h,e\rangle \hookrightarrow^* \langle s',h',\text{-}\rangle$, *then we have* $s',h' \models Post(\Phi_{po})$.

The soundness of our analysis is ensured by the soundness of the following: the entailment prover, the pure abduction mechanism, the abstract semantics (with respect to the underlying operational semantics), the pure constraint abstraction generation process, and the fixed-point calculation. Among the above, the soundness of the entailment prover and pure fixed-point calculation are already confirmed (Chin et al., 2010; Nipkow et al., 2002; Popeea and Chin, 2006), and hence we will concentrate on the soundness of abstract semantics and pure constraint abstraction derivation.

Lemma 4.3.3 (Sound pure abduction) *If* $\sigma_1 \wedge [\sigma'] \rhd \sigma_2 * \sigma_3$, *then* $\forall s,h \models Post(\sigma_1 \wedge \sigma')$, *we have* $s,h \models Post(\sigma_2 * \sigma_3)$.

Proof This is ensured by the entailment relationship in the premise of each of the pure abduction rules and the soundness of the entailment checking (Chin et al., 2010). □

Lemma 4.3.4 (Sound abstract semantics) *If* $[\![e]\!]_{\mathcal{T}}(\Delta,0) = (\Delta_1,0)$, *then for all* s,h, *if* $s,h \models Post(\Delta)$ *and* $\langle s,h,e\rangle \hookrightarrow \langle s_1,h_1,e_1\rangle$, *then there always exists* Δ_0 *such that*

$$s_1,h_1 \models Post(\Delta_0) \qquad \textit{and} \qquad [\![e_1]\!]_{\mathcal{T}}(\Delta_0,0) = (\Delta_1,0)$$

Proof The proof is done by structural induction over program constructors and is in Appendix A. □

Lemma 4.3.5 (Sound pure constraint abstraction) *Given a method with pre/-post shape templates* `pre` *and* `post`, *if our analysis successfully computes a constraint*

abstraction $\mathtt{Q}$ *in the first step without abduction, and derives a pure constraint* $\mathtt{P}$ *in the second step, then we have* $\mathtt{Q} \vdash \mathtt{post} \wedge \mathtt{P}$.

Proof This proof follows directly our procedure to compute the pure constraint abstraction from the shape one. It is also in Appendix A. □

Then based on the discussion above we have:

Theorem 4.3.6 (Soundness) *Our verification is sound with respect to the underlying operational semantics.*

4.4 Related Work

In recent years, dramatic advances have been made in automated verification of pointer safety for heap-manipulating programs. We highlight some of them here. The local shape analysis by Distefano et al. (2006) is able to infer automatically loop invariants for list-processing programs, which forms the early-version SpaceInvader tool. Gotsman et al. (2006) proposes an interprocedural shape analysis for the SLAyer tool. Berdine et al. (2007) extends the local shape analysis (Distefano et al., 2006) to handle higher-order list predicate so that more complicated real-world data structures can be analysed. Yang et al. (2008) proposs a novel abstraction operation which significantly improves the scalability of the analysis. Recently, more large industrial code can be verified by the SpaceInvader tool using the compositional analysis with bi-abductive inference (Calcagno et al., 2009; Distefano, 2009).

Several shape analyses also tried to make good use of size information. In the development of the THOR tool, Magill et al. (2007) proposes an adaptive shape analysis where additional numerical analysis can be used to help gain better precision. Its ab-

straction mechanism is also employed in C-to-gate hardware synthesis (Cook et al., 2009). Very recently, Magill et al. (2010) formulates a novel instrumentation process which inserts numerical instructions into programs, based on their shape analysis and user-provided predicates. Instrumented programs can then be used to generate pure numerical programs for further analysis. Ireland (2007) applies the symbolic evaluation technique of THOR's, and specifies the loop invariant as a combination of two parts: the shape part and the schematic content part. It can also handle the content of data structures. Different from their work, we take *both* shape and pure information into consideration when performing the abstraction, and derive the pure abstraction from the shape constraint abstraction. Our approach can be more precise as we have more information for the abstraction. Furthermore, we can directly handle data structures with stronger invariants, like sortedness and height-balanced, which have not been addressed in THOR, to the best of our knowledge. Gulwani et al. (2009) combines a set domain with its cardinality domain in a general framework. Compared with these, our approach can handle data structures with stronger invariants like sortedness, height-balanced and multiset-related invariants, which have not been addressed in the previous works. Another piece of work, by Chang et al. (2007) and Chang and Rival (2008), employs inductive checkers and checker segments to express shape and numerical information. Compared with their work, ours addresses specification refinement with pure properties (including numerical and multiset ones) in both pre- and postconditions by processing shape and pure information in two phases with the help of pure abduction. Meanwhile, our previous loop invariant synthesis (Qin et al., 2010) also infers strong loop invariants with a one-phase heavyweight abstract interpretation. Compared with this thesis, it is limited to loop analysis, whereas this thesis tackles not only loops but also methods; meanwhile this thesis is more lightweight as it solves the constraint abstraction in two phases where the second phase (pure constraint abstraction solving) utilises existing provers and is hence more modular and efficient.

There are also many other approaches to expressing heap-based domains than separation logic. Hackett and Rugina (2005) can deal with AVL-trees but is customised to handle only tree-like structures with height property. The shape analysis framework TVLA (Sagiv et al., 2002) is based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. LRP (Yorsh et al., 2006) is fully decidable over multiple linked data structures and has a finite model property. Guo et al. (2007) reports a global shape analysis that discover inductive structural shape invariants from the code. Kuncak et al. (2002) develops a role system to express and track referencing relationships among objects, where an object's role (type) depends on, and changes according to, the mutation of its referencing. Bouajjani et al. (2010) synthesises list-related invariants over infinite data domains using graph heap representation. The synthesised invariants are able to capture various aspects of data structures, such as the size, the sum or the content of linked list, relations of the data at linearly ordered or successive positions. Compared with these works, separation logic based approach benefits from the frame rule and hence supports local reasoning. Meanwhile, our approach heads towards program functional correctness including multiset-related properties, which many of previous works do not generally handle.

There are also numerous works on automated assertion discovery, for example those based on abstract interpretation (Cousot and Cousot, 1977). Compared with our work, they mainly focus on finding numerical program properties, and hence our work is complementary to theirs in the light that we also discover heap/shape information. Meanwhile, we can utilise such works as our pure solver, for example the disjunction inference (Popeea and Chin, 2006).

On the verification side, Smallfoot (Berdine et al., 2005b) is the first verification system based on separation logic. The HIP/SLEEK verification system (Nguyen et al., 2007; Nguyen and Chin, 2008) supports user-defined shape predicates over

the combined shape and numerical domain. The SLEEK tool has played a very important role in our verification. The PALE system (Möller and Schwartzbach, 2001) transforms constraints in the pointer assertion logic (PAL) into monadic second-order logic (MSO) and discharge them with MONA (Henriksen et al., 1995). It can also be used for analysis purpose once a graph type has been abstractly described with PAL. Hob (Wies et al., 2006) is a modular program verification tool for shape properties. It models relationship of objects and data structures with contents of abstract sets and uses set algebra to reason about those properties. It also allows new plug-ins to be developed to improve its power. Based on Hob, Jahob (Kuncak, 2007) takes Java as its target language and allows more general specification language with relations, specification of data structures, and combination of reasoning techniques not only at the level of modules, but also procedures, individual statements, and verification conditions. Havoc (Chatterjee et al., 2007) is another verification tool for C language about heap-allocated data structures, using a novel reachability predicate. There is another recent work on refining specifications via counterexample-guided abstraction refinement (Taghdiri, 2008) which is goal-driven and incrementally improves for given safety requirements. Among these works, our verification is distinguished because we free users from writing whole specifications by requiring only partial specifications.

4.5 Summary

We have reported in this chapter a new approach to program verification that accepts partial specifications of methods, and refines them by discovering missing constraints for numerical and multiset properties, aiming at both memory safety and functional correctness for pointer-based data structures. We employed two examples to illustrate our approach and demonstrate its viability. More detailed proof

of the feasibility of our approach can be found from the system that we built and the experimental results in Chapter 7.

Chapter 5

Synthesising Specifications for Loops/Auxiliary Methods

We have showed how our approach allows partial specifications to be given for programs to be verified and refines these specifications into more complete ones by discovering missing constraints. In this chapter, we further augment our approach to provide the user with more flexibility and automation. We propose a framework where some procedures are the main procedures of the whole program (for example program entry point with relatively simpler specifications and invoking other procedures) which are annotated with partial specifications, namely, *primary methods*. In contrast, specifications for loops and *auxiliary methods* (which are invoked by the primary methods) can then be systematically discovered by our augmented mechanism, with the help of information propagated from the primary methods. This mechanism brings more agility to our verification and may reduce users' annotation to a further extent.

5.1 Introduction

In the previous chapter, we demonstrated an approach to the reduction of user-supplied specifications for program verification, which only requires users to provide annotations expressing shape information of programs' input and output, and will help users to refine such specifications with inferred obligations for numerical and/or content constraints, such that the shape-only specifications become sound. However, a direct question to this approach is: is there any opportunity for the specifications to be reduced further?

This question may be originated from the requirement of our verification, which runs in a modular way by refining the partial specifications for *each* method. Therefore, to verify a whole program, users still need to annotate every procedure with its shape specifications. Sometimes this could be difficult for the users: they might be more familiar with some of the procedures which are main methods to implement certain functions, but less knowledgeable about some other procedures which are invoked by those main methods to perform auxiliary functions. Meanwhile, it is quite often that such procedures for auxiliary functions have sophisticated annotations, while the main procedures, in contrast, come up with relatively simple specifications. For example, this situation applies to the insertion method of an AVL-tree, which has several procedures for tree rotations with quite complex specifications.

As an answer to this question, we propose a solution by dividing the procedures in a program into two types: primary procedures which are the main/entry methods to implement some functions, and auxiliary ones invoked by the primary procedures. As an instance, the `insert_sort` procedure in the previous chapter (page 66) can be regarded as a primary procedure, and its callee `insert` as an auxiliary one. In this setting, we allow the user to annotate only the primary procedures with shape-

based specifications, and leave blank for the auxiliary ones.[1] For the procedures marked as primary by the user, we still employ previous chapter's algorithm to infer their missing constraints. For the auxiliary ones, as the user does not provide a pair of (shape-only) specifications, we will first synthesise their specifications of shape information based on its calling context in the caller procedure, and utilise the same algorithm presented in the previous chapter to obtain the remaining constraints. This approach may increase the time-consumption of verification (as it has to synthesise the shape specifications) and lose some modularity (as such synthesis requires calling context information from the caller during the verification of callee; we denote it as *semi-modular* since it is still modular if we regard a primary method and its affiliating auxiliary methods as a whole integrity). However, at such expense we can further reduce users' workload of marking annotations and provide them with more flexibility as they can omit the auxiliary procedures' specifications if they want to.

To summarise, our division of auxiliary methods from primary ones and synthesis of auxiliary methods' specifications represent one more step towards an automated verification of heap-manipulating programs. It applies more program analysis techniques in the verification process and offers more agility to end-users by enabling them to choose primary methods to annotate and auxiliary methods to leave for computers. By such a choice they can further balance the trade-off between human intelligence's assistance to the verification and its level of automation. This framework has the following characteristics:

- *Further reduction of user annotations*: As it is not necessary to annotate the auxiliary methods, user annotations are further reduced compared with the

[1] The user can also choose to annotate an auxiliary method for time-saving or precision purposes; in such a situation this approach degenerates into the one in the previous chapter.

results from the last chapter. This will also be exhibited by our experiments in subsequent chapters.

- *Flexibility*: Apart from allowing users to define their own predicates, we enable them to decide which methods to annotate and which not. They are now more involved in the verification process, because if they are dissatisfied by the time cost or precision achieved of some automatically calculated specifications, then they can specify those by themselves; and vice versa.

- *Semi-automation*: We classify our approach as semi-automatic, because the user is allowed to interfere and guide the verification at any point, which is linked to the previous feature.

The rest of this chapter is organised as follows. Section 5.2 illustrates how our improved method works for the motivating examples in the last chapter. Section 5.3 revises our programming language according to this improvement. Then Section 5.4 formulates how such synthesis of specifications for auxiliary methods is performed. The last section will discuss about related works and conclude this chapter.

5.2 The Approach

This section reviews the two illustrative examples in Section 4.2.1 to describe our approach informally.

5.2.1 First Illustrative Example Revisited

We first illustrate our approach of specification synthesis using method `insert` in Figure 5.1. We show how our synthesis infers missing specifications for auxiliary methods so that the refinement of both primary and auxiliary methods' specifications may continue.

Most of the program in Figure 5.1 is identical as the one in the last chapter, including all the code and the shape-only specification provided for procedure `insert_sort`. The focus is now on `insert`, which inserts a node `x` into a sorted list `r`. It judges three cases and has a non-tail-recursive call to itself in the last case (to insert `x` after list `r`'s head). It could be noticed that now the formerly existing partial specification for `insert` is gone.

As can be imagined from the fact above, currently the program is divided into two parts: the primary (entry) procedure for the implementation of the algorithm, i.e. `insert_sort`, and the auxiliary (invoked) procedure `insert`. For these two procedures, we suppose that the user chooses not to provide a specification for the auxiliary one, and hence we must synthesise its specifications so that the refinement of its specifications can continue.

To achieve this objective, we alter slightly the way in which we verify the whole program. Previously we first refine the shape specification for `insert` (when `insert` has its user-supplied annotation) to obtain a completed specification for it, and apply such specification in the verification of `insert_sort` (in line 8 of the program). Now as we do not have `insert`'s annotation, we firstly begin with the verification of `insert_sort` since it has shape specifications to start with. Then, when the forward analysis reaches line 8 to invoke `insert`, we assume that the current program

```
 1 class Node { int val; Node next; }
 2 Node insert_sort(Node x)
 3  requires x::llB⟨S⟩
 4  ensures  res::sllB⟨T⟩ {
 5   if (x.next == null) return x;
 6   else { Node s = x.next;
 7     Node r = insert_sort(s);
 8     return insert(r, x);
 9   }
10 }
11 Node insert(Node r, Node x)
12   if (r == null) {
13     x.next = null; return x;
14   } else if (x.val <= r.val) {
15     x.next = r; return x;
16   } else {
17     r.next = insert(r.next, x);
18     return r;
19   }
20 }
```

Figure 5.1: The insertion sort program for singly linked lists.

state satisfies its precondition, and try to discover its pre-shape accordingly. For `insert`'s post-shape, we conduct another analysis over its procedure body to synthesise it. Finally, when both pre-shape and post-shape are ready for `insert`, we

exploit the means in Chapter 4 to refine its specification with (possible) extra quantitative/multiset constraints, and use this specification so as to continue to verify `insert_sort`.[2]

Let us roll back to the forward analysis of `insert_sort`. When it approaches line 8, the abstract program state at the call site is

$$\texttt{x::Node}\langle \texttt{v,s}\rangle * \texttt{r::sllB}\langle \texttt{S}_\texttt{r}\rangle$$

Then `insert` should be invoked; however we do not know its specifications. As aforementioned, the pre-shape is directly synthesised from the abstract program state at the call site. To synthesise post-shapes, we unroll the recursive call once, symbolically execute the unrolled method body (starting from the pre-shape) to obtain a post-state, and then use the post-state to filter out any invalid post-shapes from the set of possible post-shapes (drawn from all available shape predicates). For this example, the possible post-shape candidates can be (a) $\texttt{x::sllB}\langle \texttt{S}_1\rangle * \texttt{res::sllB}\langle \texttt{S}_2\rangle$, and (b) $\texttt{res::sllB}\langle \texttt{S}\rangle$, etc. The symbolic execution gives the following post-state:

$$\begin{array}{l}\texttt{x::Node}\langle \texttt{v}, \texttt{null}\rangle \wedge \texttt{x=res} \vee \\ \quad \texttt{x::Node}\langle \texttt{v}, \texttt{r}\rangle * \texttt{r::sllB}\langle \texttt{S}_1\rangle \wedge \texttt{x=res} \wedge (\forall \texttt{u} \in \texttt{S}_1 \cdot \texttt{v} \leq \texttt{u}) \vee \\ \quad\quad \texttt{r::Node}\langle \texttt{u}, \texttt{x}\rangle * \texttt{x::Node}\langle \texttt{v}, \texttt{null}\rangle \wedge \texttt{r=res} \wedge \texttt{u} \leq \texttt{v} \vee \\ \quad\quad\quad \texttt{r::Node}\langle \texttt{u}, \texttt{x}\rangle * \texttt{x::Node}\langle \texttt{v}, \texttt{r}_1\rangle * \texttt{r}_1\texttt{::sllB}\langle \texttt{S}_1\rangle \wedge \texttt{r=res} \wedge \texttt{u} \leq \texttt{v} \wedge (\forall \texttt{w} \in \texttt{S}_1 \cdot \texttt{v} \leq \texttt{w})\end{array}$$

which does not entail the candidate (a), so we filter it out. Taking (b) as the post-shape, we now have a shape specification for `insert`:

$$\texttt{requires r::sllB}\langle \texttt{S}\rangle * \texttt{x::Node}\langle \texttt{v}, \_\rangle \;\; \texttt{ensures res::sllB}\langle \texttt{T}\rangle$$

Then we can employ the same refinement process for primary procedures to obtain the specification

$$\texttt{requires r::sllB}\langle \texttt{S}\rangle * \texttt{x::Node}\langle \texttt{v}, \_\rangle \;\; \texttt{ensures res::sllB}\langle \texttt{T}\rangle \wedge \texttt{T=S} \sqcup \{\texttt{v}\}$$

[2]Note that here we actually pay more attention to the postcondition of the specification, as it is more important for us to proceed with the remaining verification.

for `insert` and continue with the verification of `insert_sort`.

5.2.2 Second Illustrative Example Revisited

Now we prove that our approach is also feasible for the second example in the previous chapter.

The code we verify now is in Figure 5.2. In this example there is only one procedure to verify; therefore it is the primary procedure with shape annotations. The auxiliary procedure in this scenario is the while loop in lines 7-13. Compared with the previous example in Figure 4.2, we note that the very sophisticated annotation for the while loop $\texttt{head}::\texttt{sdlB}\langle \texttt{p}, \texttt{q}, \texttt{S}\rangle \mathrel{-\!\!*\!\!\!\rightarrow} \texttt{head}::\texttt{sdlB}\langle \texttt{p}_\texttt{h}, \texttt{q}_\texttt{h}, \texttt{S}_\texttt{h}\rangle * \texttt{root}::\texttt{sdlB}\langle \texttt{p}_\texttt{r}, \texttt{q}_\texttt{r}, \texttt{S}_\texttt{r}\rangle * \texttt{end}::\texttt{sdlB}\langle \texttt{p}_\texttt{e}, \texttt{q}_\texttt{e}, \texttt{S}_\texttt{e}\rangle$ has already gone. This fact again signifies the importance to eliminate auxiliary procedures' annotations, because such procedures often help the primary ones handle auxiliary tasks with complicated input/output, and accordingly their annotations are usually more complex than the primary procedures' counterparts.

The task of this while loop is to discover the centre node of the given list segment referenced by `head`. It traverses the list segment with two pointers `root` and `end`. The `end` pointer goes towards the list segment's tail twice as fast as `root`. When `end` arrives at the tail of the segment (`tail`), `root` will point to the list segment's centre node.

We utilise the same framework as in the last example to verify the whole program. Firstly the verification begins with the outer primary procedure `sdl2nbt` until the loop at line 7. This time it does not have user annotations; as aforementioned, our verification must first synthesise its pre- and post-states with shape information,

```
 0 class Node2 { int val; Node2 prev;
                 Node2 next; }
 1 Node2 sdl2nbt(Node2 head, Node2 tail)
 2  requires head::sdlB⟨p, q, S⟩
 3  ensures  res::nbt⟨S_res⟩
 4 {
 5  Node2 root = head;
 6  Node2 end = head;
 7  while(end != tail) {
 8    end = end.next;
 9    if (end != tail) {
10      end = end.next;
11      root = root.next;
12    }
13  }
14  if (head == root)
15    root.prev = null;
16  else
17    root.prev = sdl2nbt(head, root);
18  Node2 tmp = root.next;
19  if (tmp == tail)
20    root.next = null;
21  else {
22    tmp.prev = null;
23    root.next = sdl2nbt(tmp, tail);
24  }
25  return root;
26 }
```

Figure 5.2: Algorithm to convert a sorted doubly-linked list to a node-balanced tree.

and then proceed with its constraint abstraction. For pre-shape it is straightforward as the program state before the loop will provide relevant shape information. For post it is done by checking the loop body (unrolled once)'s symbolic execution result against all possible abstracted shapes. For the previous example, we first generate all possible shapes according to the variables accessed by the loop, such as (a) $\texttt{head::sdlB}\langle \texttt{p}_\texttt{h}, \texttt{q}_\texttt{h}, \texttt{S}_\texttt{h}\rangle * \texttt{root::sdlB}\langle \texttt{p}_\texttt{r}, \texttt{q}_\texttt{r}, \texttt{S}_\texttt{r}\rangle$, and (b) $\texttt{head::sdlB}\langle \texttt{p}_\texttt{h}, \texttt{q}_\texttt{h}, \texttt{S}_\texttt{h}\rangle * \texttt{root::nbt}\langle \texttt{h}_\texttt{r}, \texttt{b}_\texttt{r}, \texttt{S}_\texttt{r}\rangle$, and many so forth. Then the unrolled loop body is symbolically executed several times to filter out any invalid shape as an invariant. In the example's case, executing the loop body will yield the following result:

$$\begin{array}{l}\texttt{head::Node2}\langle \texttt{v}, \texttt{p}, \texttt{end}\rangle \wedge \texttt{head=root} \wedge \texttt{end=tail} \vee \\ \quad \texttt{head::Node2}\langle \texttt{v}_\texttt{h}, \texttt{p}, \texttt{root}\rangle * \texttt{root::Node2}\langle \texttt{v}_\texttt{r}, \texttt{head}, \texttt{end}\rangle \wedge \texttt{end=tail}\end{array} \tag{5.1}$$

where (b) is directly filtered out since (5.1) $\vdash (b) * \texttt{true}$ fails. However (a) remains a candidate, as both (5.1) $\vdash (a) * \texttt{true}$ holds. Therefore, regarding (a) as a possible shape post, we can employ the same approach for the whole method to generate a constraint abstraction for the while loop, and solve it to obtain its postcondition

$$\begin{array}{l}\texttt{head::sdlB}\langle \texttt{null}, \texttt{root}, \texttt{S}_\texttt{h}\rangle * \texttt{root::sdlB}\langle \texttt{p}, \texttt{tail}, \texttt{S}_\texttt{r}\rangle \wedge \\ \quad \texttt{end=tail} \wedge \texttt{S=S}_\texttt{h} \sqcup \texttt{S}_\texttt{r} \wedge (\forall \texttt{x} \in \texttt{S}_\texttt{h}, \texttt{y} \in \texttt{S}_\texttt{r} \cdot \texttt{x} \leq \texttt{y}) \wedge 0 \leq |\texttt{S}_\texttt{r}| - |\texttt{S}_\texttt{h}| \leq 1\end{array}$$

to continue with the verification.

One more note for the while loop in this example is that the symbolic execution may actually permit more than one shapes to enter as candidates, for instance, $\texttt{head::sdlB}\langle \texttt{p}_\texttt{h}, \texttt{q}_\texttt{h}, \texttt{S}_\texttt{h}\rangle$. Generally this does not affect the analysis result, as we allow the analysis to continue with all possible postconditions computed from this while loop, and always choose the most precise final result. In the motivating example, both $\texttt{head::sdlB}\langle \texttt{p}_\texttt{h}, \texttt{q}_\texttt{h}, \texttt{S}_\texttt{h}\rangle$ and (a) are valid shape postconditions for the loop, but later the former one will cause the analysis to fail in line 15/17, because it inappropriately approximated the invariant and hence lost information about `root`. Since we synthesise all possible shapes, we can always select those shapes sufficiently strong to support further analysis to obtain a meaningful result.

5.3 Programming Language

To cater for our specification synthesis approach, we have revised the programming language definition slightly. As shown in Figure 5.3, the language now allows both primary procedures and auxiliary procedures. Their status is decided implicitly by the specifications they have: the procedures with shape specifications are primary procedures, and the ones without any specification are auxiliary ones (in need of synthesis).

5.4 The Verification

This section formulates our whole framework to verify a program consisting of both primary and auxiliary methods, as well as the algorithms for shape specification synthesis.

5.4.1 The Overall Approach

Generally speaking, our whole framework of verification is still founded on the generation of constraint abstractions from the program being verified and also solving such constraint abstractions. For the primary methods, we expect the user to provide shape information in both pre- and postconditions to help the constraint abstraction generation. For loops and auxiliary methods, we produce a set of candidate abstractions for their specifications from both their program body and the current state to invoke them, such that their corresponding constraint abstractions can be built and solved in a same way as primary methods. The overall algorithm is listed in Figure 5.4.

Program	*Prog*	::=	$\boldsymbol{tdecl\ meth}$	
Type declaration	*tdecl*	::=	*classt* | *spred* | *lemma*	
Class declaration	*classt*	::=	`class` c { $\boldsymbol{field}$ }	
Field declaration	*field*	::=	$t\ v$	
Type	t	::=	$c \mid \tau$	
Procedure declaration	*meth*	::=	$t\ mn\ ((\boldsymbol{t\ v}); (\boldsymbol{t\ v}))\ \{e\}$	
		|	$t\ mn\ ((\boldsymbol{t\ v}); (\boldsymbol{t\ v}))\ \boldsymbol{mspec}\ \{e\}$	
Built-in type	τ	::=	`int` | `bool` | `void`	
Expression	e	::=	d	heap-insensitive atomic
		|	$d[v]$	heap-sensitive atomic
		|	$v{=}e$	assignment
		|	$e_1; e_2$	sequence
		|	$t\ v;\ e$	local variable
		|	`if` $(v)\ e_1$ `else` e_2	
		|	`while` $v\ \{e\}$	
Heap-insensitive atomic	d	::=	-	skip
		|	`null`	null reference
		|	k^τ	constant
		|	v	variable
		|	`new` $c(\boldsymbol{v})$	allocation
		|	$mn(\boldsymbol{u}; \boldsymbol{v})$	method call
Heap-sensitive atomic	$d[v]$	::=	$v.f$	field read
		|	$v.f{=}w$	field write
		|	`free`(v)	deallocation

Figure 5.3: The programming language for the specification synthesis framework.

Algorithm $\mathsf{Verify}(\mathcal{T}, \mathcal{S}, mn, \sigma, \boldsymbol{x}, \boldsymbol{y})$

1 **case** mn **of**

2 | $\texttt{while}\ (w)\ \{e_0\} \rightarrow f := \mathsf{fresh\_name}();\ e := \texttt{if}\ (w)\ \{e_0;\ f(\boldsymbol{x}; \boldsymbol{y})\};$
$(\boldsymbol{u}, \boldsymbol{v}) := (\boldsymbol{x}, \boldsymbol{y});\ ([(\Phi^i_{pr}, \Phi^i_{po})], n) := \mathsf{Preproc}(\mathcal{T}, \mathcal{S}, f, \boldsymbol{x}, \boldsymbol{y}, e_0, \sigma, \boldsymbol{x}, \boldsymbol{y});$
$prim := \texttt{false};$

3 | $t\ mn\ ((\boldsymbol{t}\ \boldsymbol{u_0}); (\boldsymbol{t}\ \boldsymbol{v_0}))\ \{e_0\} \rightarrow f := mn;\ e := e_0;\ (\boldsymbol{u}, \boldsymbol{v}) := (\boldsymbol{u_0}, \boldsymbol{v_0});$
$([(\Phi^i_{pr}, \Phi^i_{po})], n) := \mathsf{Preproc}(\mathcal{T}, \mathcal{S}, f, \boldsymbol{u}, \boldsymbol{v}, e_0, \sigma, \boldsymbol{x}, \boldsymbol{y});\ prim := \texttt{false};$

4 | $t\ mn\ ((\boldsymbol{t}\ \boldsymbol{u_0}); (\boldsymbol{t}\ \boldsymbol{v_0}))\ (\mathit{requires}\ \Phi^{pr}_i\ \mathit{ensures}\ \Phi^{po}_i)^m_{i=1}\ \{e_0\} \rightarrow f := mn;$
$e := e_0;\ (\boldsymbol{u}, \boldsymbol{v}) := (\boldsymbol{u_0}, \boldsymbol{v_0});\ n := m;\ (\Phi^i_{pr}, \Phi^i_{po})^m_{i=1} := (\Phi^{pr}_i, \Phi^{po}_i)^m_{i=1};$
$prim := \texttt{true};$

5 **end case**

6 $sps := \emptyset$

7 **for** $i := 1$ **to** n **do**

8 $sp := \mathsf{CA\_Gen\_Solve}(\mathcal{T}, f, e, \Phi^i_{pr}, \Phi^i_{po}, \boldsymbol{u}, \boldsymbol{v})$

9 **if** $prim = \texttt{false}$ **and** $sp \neq \mathsf{fail}$ **then return** (f, sp)

10 **else if** $prim = \texttt{true}$ **then** $sps := sps \cup sp$

11 **end if**

12 **end for**

13 **return** (f, sps)

end Algorithm

Figure 5.4: Main verification algorithm.

Our verification algorithm takes as input all available specifications and shapes, and the code segment to be verified, together with an optional conjunctive program state and two variable sequences (mainly for loops and auxiliary procedures). It runs in two steps. The first step recognises whether the procedure is transformed from a loop (line 2) or an auxiliary one (line 3), or a primary procedure with user-supplied shape specifications (line 4). If it is a loop originally or an auxiliary procedure, then it will undergo a pre-processing step (Figure 5.5) which discovers a list of candidate shape specifications for that procedure.

Algorithm Preproc$(\mathcal{T}, \mathcal{S}, f, \boldsymbol{u}, \boldsymbol{v}, e, \sigma, \boldsymbol{x}, \boldsymbol{y})$

1 $sps := [\,]$;

2 $prs :=$ SynPre$(\mathcal{S}, f, \boldsymbol{u}, \boldsymbol{v}, \sigma, \boldsymbol{x}, \boldsymbol{y})$

3 **for** $\Phi_{pr} \in prs$ **do**

4 $pos :=$ SynPost$(\mathcal{T}, \mathcal{S}, f, e, \Phi_{pr}, \boldsymbol{u}, \boldsymbol{v})$

5 $sps :=$ concat(sps, pos)

6 **end for**

7 **return** $(sps, |sps|)$

end Algorithm

Figure 5.5: Pre-processing algorithm.

The pre-processing algorithm Preproc in Figure 5.5 mainly invokes the shape synthesis procedures to discover all possible pre- and post-shapes for loops and auxiliary procedures, as shown in lines 1 and 4. Then the list of shape pairs (specifications) are returned and used in further analysis. The details of shape synthesis algorithms will be introduced in Section 5.4.2.

After the first step, the procedure to be verified is guaranteed with some shape

specifications. Then the second step will generate constraint abstractions and solve them according to the shapes given in the specifications (at line 8 and described in detail in Section 4.3.1). The solutions are then used to refine the shape specifications with pure constraints.

For the auxiliary procedures and loops, we apply a lazy scheme: as the pre-processing may yield several possible shape specifications in a list (ordered with heuristics such that the specifications with more possibility to make the whole verification succeed are closer to the list head), we try to verify each in sequence. Once a specification can be verified against the program, then it is returned and the other ones are omitted. This is reasonable as our main purpose is to verify and refine the specification for the primary procedure, and thus we view the specifications gained for the auxiliary procedures as affiliated results. In this way we try to make our verification more scalable, as will be described in later sections.

5.4.2 Specification Synthesis for Auxiliary Methods and Loops

For auxiliary procedures, as we do not expect the user to provide specification annotations, we conduct a pre-analysis (Figure 5.6 and Figure 5.7) to synthesise the pre- and post-shapes before invoking the constraint abstraction generation algorithm (Figure 4.4). Loops are dealt with by analysing their tail-recursive versions in the same way.

The pre-shape synthesis algorithm SynPre (Figure 5.6) takes in as input the set of shape predicates ($\mathcal{S}$), the auxiliary method name (f), its formal parameters ($\boldsymbol{u}, \boldsymbol{v}$), the current symbolic state in which f is called (σ), and the corresponding actual parameters ($\boldsymbol{x}, \boldsymbol{y}$) of the invocation. The algorithm first obtains possible shape

Algorithm SynPre($\mathcal{S}, f, \boldsymbol{u}, \boldsymbol{v}, \sigma, \boldsymbol{x}, \boldsymbol{y}$)

1 $C :=$ ShpCand($\mathcal{S}, \boldsymbol{u}, \boldsymbol{v}$)

2 **for** $\sigma_C \in C$ **do**

3 **if** $\sigma \not\vdash [\boldsymbol{x}/\boldsymbol{u}, \boldsymbol{y}/\boldsymbol{v}]\sigma_C$

4 **then** $C := C \backslash \{\sigma_C\}$

5 **end if**

6 **end for**

7 **return** C

end Algorithm

Figure 5.6: Precondition synthesis algorithm.

candidates from the parameters $\boldsymbol{u}, \boldsymbol{v}$ with ShpCand (line 1), tests for each shape whether it is a sound abstraction for the method's pre-shape with entailment (line 3), then picks up a sound abstraction for the method's pre-shape with entailment, and filter out the ones which fail (line 4). Finally the pre-shape abstraction is returned. While we use an enumeration strategy here, the number of possible shape candidates per type is small as it is strictly limited by what the user provides in the primary methods, and further filtered and prioritised by our system.

To synthesise post-shapes (SynPost in Figure 5.7), we also assign C as possible shape candidates (line 1). We unroll f's body e once (i.e. replace recursive calls to f in e with a substituted e) and symbolically execute it with the algorithm in Figure 4.5 (line 3), assuming f has a specification *requires* Φ_{pr} *ensures* `false` (line 2). The postcondition `false` is used to ensure that the execution only considers the effect of the program branches with no recursive calls (to f itself). We then use Δ to find out appropriate abstraction of post-shape (line 5), which is paired with Φ_{pr} and returned

Algorithm SynPost $(\mathcal{T}, \mathcal{S}, f, e, \Phi_{pr}, \boldsymbol{u}, \boldsymbol{v})$

1 $C := \mathsf{ShpCand}(\mathcal{S}, \boldsymbol{u}, \boldsymbol{v})$

2 $\mathcal{T}' := \mathcal{T} \cup \{f(\boldsymbol{u}, \boldsymbol{v})$ *requires* Φ_{pr} *ensures* `false` $\{e\}\}$

3 $\Delta := \mathsf{Symb\_Exec}(\mathcal{T}', f, \mathsf{syn\_unroll}(f, e), \Phi_{pr})$

4 **for** $\sigma_C \in C$ **do**

5 **if** $\Delta \wedge [\sigma] \not\models \sigma_C$ **then** $C := C \backslash \{\sigma_C\}$ **end if**

6 **end for**

7 **return** $\mathsf{pair\_spec\_list}(\Phi_{pr}, C)$

end Algorithm

Figure 5.7: Postcondition synthesis algorithm.

as result. The function pair_spec_list forms an ordered list of pre-/post-shape pairs, each of which has Φ_{pr} as pre-shape and a Φ_{po} in C as post-shape.

As can be seen, the generation of possible shape candidates plays an important role in our synthesis. Its implementation is a recursive algorithm, each recursion of which decides whether or not to include a given variable and its shape in the produced separation conjunction as result. The algorithm is listed in Figure 5.8.

This algorithm is the foundation of ShpCand. It invokes length to obtain the length of a list, and $\mathsf{isCompatible}(v, S)$ is a type checker to test whether variable v's type is consistent with the shape predicate S, namely, whether the recursive branches of S's definition look like $\texttt{root}::c\langle\boldsymbol{v}\rangle * \ldots$ where c is the type of v. Therefore, based on ShpCandRec, the definition of ShpCand is straightforward as follows:

$$\mathsf{ShpCand}(\mathcal{S}, \boldsymbol{u}, \boldsymbol{v}) =_{df} \mathsf{ShpCandRec}(\mathcal{S}, \mathsf{concat}(\boldsymbol{u}, \boldsymbol{v}), 1)$$

Now we illustrate ShpCand with an example. If we have two parameters `x` and `y`

Algorithm ShpCandRec $(\mathcal{S}, \boldsymbol{v}, d)$

```
1   R := ∅
2   if d = length(v) then
3     for S ∈ 𝒮 do
4       if isCompatible(v_d, S) then
5         R := R∪{v_d::S⟨u⟩}, where u are fresh logical variables
6       end if
7       R := R∪{emp}
8     end for
9   else
10    R_0 := ShpCandRec(𝒮, v, d+1)
11    for σ ∈ R_0 do
12      for S ∈ 𝒮 do
13        if isCompatible(v_d, S) then
14          R := R∪{v_d::S⟨u⟩ ∗ σ}, where u are fresh logical variables
15        end if
16      end for
17      R := R∪{σ}
18    end for
19  end if
20  return R
end Algorithm
```

Figure 5.8: Shape candidate generation algorithm.

with type `Node`, and the user has defined two shape predicates `llB` and `sllB` with `Node`, then the list of all possible shape candidates for the two variables (C) will be [x::sllB⟨S⟩ ∗ y::sllB⟨T⟩, x::llB⟨S⟩ ∗ y::sllB⟨T⟩, x::sllB⟨S⟩ ∗ y::llB⟨T⟩, x::llB⟨S⟩ ∗ y::llB⟨T⟩, x::sllB⟨S⟩, y::sllB⟨S⟩, x::llB⟨S⟩, y::llB⟨S⟩, emp]. Then elements of this list will be checked against appropriate abstract states (line 3 in Figure 5.6 and line 5 in Figure 5.7) where most elements should be reduced because they are not sound abstractions. For example, in the previous list, only x::llB⟨S⟩ ∗ y::llB⟨T⟩ remains in the list and participates in further verification. Meanwhile, for any candidate variable, ShpCand only picks up *compatible* shape predicates from $\mathcal{S}$, which reduces more shape candidates. For instance, if the data structure manipulated by the method is of type `Node`, then ShpCand rules out shape predicates specifying other types of data structures, for example doubly-linked lists and trees, etc.

Our shape synthesis generally keeps only highly relevant abstractions. For the while loop in Section 5.2.2, we filtered out 24 (of 26) abstractions. Generally, in case that there are several abstractions as candidate specifications, we employ some other mechanisms to reduce them further. Firstly, we prioritise post-shapes with same (or stronger) predicates as in precondition since it is more likely that the output will have the same or similar shape predicates as the input, e.g. `x` is expected to remain as `sllB` (or stronger) if it points to `sllB` as input. Secondly, we employ a lazy scheme when refining the synthesised pre/post-shapes (to complete specifications). We retrieve (and remove) the pre/post-shape pair from the head of the list, (1) use the refinement algorithm (Figure 4.4) to obtain a specification for the auxiliary method, and (2) continue the analysis for the primary method. If the analysis for the primary method succeeds, we will ignore all other synthesised pre/post-shapes from the list. If either (1) or (2) fails, we will try the next one from the list. Note that our synthesis of shape specification could only cater to one predicate per parameter/result. In cases where more complex shape specifications are needed, we allow users to specify them directly for the respective auxiliary method. These

mechanisms help to keep attempts over candidate specifications at a minimum level.

A final note on this synthesis approach is on the fact that it carries on with primary procedure's verification until the call site of an auxiliary procedure, and then verifies the auxiliary procedure before completing the verification for the primary procedure. Therefore, it is at the expense of loss of modularity, if we consider both primary and auxiliary procedures as having equivalent status (both are procedures of the program). However, from a pragmatic perspective, we tend to regard this approach as *semi-modular*, since if we take every primary procedure and its affiliated auxiliary procedures as an integrated part, then this approach is modular in this sense. As long as such "integrated parts" in programs are not oversized, it will not affect the scalability of our approach.

5.4.3 Revised Symbolic Execution Rules

Applying the framework described in this chapter, we need to make some slight alterations to the symbolic execution rules when we invoke the algorithm in Figure 4.5 (line 3) to cater for the newly added feature of auxiliary procedures. The added rules are as follows:

$$\frac{(\boldsymbol{x},\boldsymbol{y}) = \mathsf{vars}(w,e) \quad (f,\mathcal{T}_1) = \mathsf{Verify}(\mathcal{T},\mathcal{S},\texttt{while}(w)\{e\},\sigma,\boldsymbol{x},\boldsymbol{y}) \quad \mathcal{T}' = \mathcal{T}\cup\mathcal{T}_1}{\mathsf{exec}(\texttt{while}(w)\{e\})(\mathcal{T})(\sigma,0) \rightsquigarrow \mathsf{exec}(f(\boldsymbol{x};\boldsymbol{y}))(\mathcal{T}')(\sigma,0)}$$

$$\frac{t\ mn\ ((t_i\ u_i)_{i=1}^{m};(t_i\ v_i)_{i=1}^{n}) \notin \mathcal{T} \quad (f,\mathcal{T}_1) = \mathsf{Verify}(\mathcal{T},\mathcal{S},mn,\sigma,\boldsymbol{x},\boldsymbol{y}) \quad \mathcal{T}' = \mathcal{T}\cup\mathcal{T}_1}{\mathsf{exec}(mn(x_1..x_m;y_1..y_n))(\mathcal{T})(\sigma,0) \rightsquigarrow \mathsf{exec}(mn(x_1..x_m;y_1..y_n))(\mathcal{T}')(\sigma,0)}$$

where the set $\mathcal{S}$ is supposed to contain all the shape definitions provided by user. As can be seen, these two new rules are used for the invocation of a while loop or an auxiliary method which has not been verified, where we employ the verification algorithm in Figure 5.4 recursively to obtain its postcondition. Therefore we

know that when the verification over a procedure's body meets an invocation to a loop/auxiliary procedure, these two rules will be triggered to fill in the specifications of the loop/auxiliary procedure, allowing the verification of its caller to carry on.

5.4.4 Soundness

The approach presented in this chapter is essentially an extension of the one in the previous chapter. Its soundness definition also follows the previous one, referring to the underlying operational semantics:

Definition 5.4.1 (Soundness) *For a method definition* $t\ mn\ ((\boldsymbol{t\ u}); (\boldsymbol{t\ v}))\ \{e\}$, *if our analysis synthesises its specification as* $t\ mn\ ((\boldsymbol{t\ u}); (\boldsymbol{t\ v}))$ *requires* Φ_{pr} *ensures* $\Phi_{po}\ \{e\}$, *then for all* $s, h \models Post(\Phi_{pr})$, *if* $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', \text{-}\rangle$, *then we have* $s', h' \models Post(\Phi_{po})$.

Therefore the soundness of the whole approach can be reduced to the soundness of our synthesis of shape specifications. To prove this, we need to review our pre-condition/postcondition synthesis algorithms. From these two algorithms, we can see that our synthesised pre-shape must satisfy the abstract state at the calling context (because of the entailment relationship), and the post-shape is checked to see whether it could possibly be abstracted as the execution result of the unfolded program. From the soundness of entailment checking and abduction, we have

Theorem 5.4.2 (Soundness) *Our verification is still sound with respect to the underlying operational semantics, with the specification synthesis mechanism added.*

Detailed proof can be found in Appendix A.

5.5 Related Work

The verification framework in this chapter, which divides the program into primary and auxiliary procedures, applies some program analysis techniques to synthesise raw specifications for auxiliary procedures and complete them. Its counterparts in the state-of-the-art are introduced in the previous chapter in several categories. The shape-only analyses include Distefano et al. (2006); Gotsman et al. (2006); Berdine et al. (2007); Yang et al. (2008); Calcagno et al. (2009). Some works capable of handling numerical or content information include Magill et al. (2007, 2008, 2010); Ireland (2007); Maclean et al. (2009); Gulwani et al. (2009); Chang et al. (2007); Chang and Rival (2008). On the contrary to these works, we only apply program analysis techniques over loops/auxiliary procedures, and we do not attempt to compute a fixed-point for their postcondition (instead we just unroll the procedure body once to execute it symbolically, resulting in a sound approximation of the post-shape). Our approach runs in this way mainly because we want to minimise the cost of such synthesis, compared with their relatively high cost of fixed-point iteration. A potential expense of this choice could be a coarsely generated post-shape as the synthesis result; however such expense will be digested by our specification refinement introduced in Chapter 4. Meanwhile, it is possible for us to employ fixed-point calculation in the synthesis of post-shape; yet it is an orthogonal problem to this approach as addressed in our other works (Luo et al., 2010b; Qin et al., 2010).

On the verification side, compared with the previous works (Berdine et al., 2005b; Nguyen et al., 2007; Wies et al., 2006; Kuncak, 2007; Chatterjee et al., 2007; Taghdiri, 2008), our verification is now even more distinguished than in the previous chapter because we further free users from writing specifications for loops and auxiliary methods and we will discover these annotations for them.

5.6 Summary

In this chapter we augment our proposed approach to the refinement of specifications by requiring only partial specification for primary procedures. Specifications for loops and auxiliary procedures can then be systematically discovered, with the help of information propagated from the primary methods. On the basis of this technique we have slightly altered our framework of verification which now provides more flexibility for the end user.

Chapter 6

Verifying Programs with Unknown Components

Verification of programs with invocations to unknown components is a practical problem, because in many scenarios not all code of programs to be verified is available. Those unknown components also pose a challenge for their verification. This chapter addresses this problem with an attempt to verify both memory safety and functional correctness of such programs using pointer-based data structures. Provided with a Hoare-style specification $\{\Phi_{pr}\}$ prog $\{\Phi_{po}\}$ where program prog contains calls to some unknown components unknown, we infer a specification $mspec_u$ for unknown from the calling contexts, such that the problem of verifying prog can be safely reduced to the problem of proving that unknown (once its code is available) meets the derived specification $mspec_u$. The expected specification $mspec_u$ for unknown is automatically calculated using an abduction-based shape analysis specifically designed for our combined abstract domain.

6.1 Introduction

A recent prevalent trend of component-based software engineering (Kozaczynski and Booch, 1998) poses great challenge for quality assurance and verification of programs. This methodology involves the integration of software components from both native development and third-parties, and thus the source code of some components/procedures might be unknown for verification. This problem is quite practical and has multiple forms in various scenarios. For example, some programs may have calls to third-party library procedures whose code is not accessible (e.g. in binary form). Some components may be invoked by remote procedure calls only with a native interface such as COM/DCOM (Sessions, 1998). Still, some components could be used for dynamic upgrading of running systems whose cost of being stopped/restarted is too expensive to bear (Szyperski, 2003). Other scenarios include function pointers (e.g. in C), interface method invocation (e.g. in OO) and mobile code, which all contain procedures not available for static verification.

To verify such programs, existing approaches generally do not provide elegant solutions:

- Black-box testing (Beizer, 1996) regards the unknown components as black-boxes to test their functionality, which cannot formally prove the absence of program bugs. Especially in safety-critical systems a bug failed to be found by testing may cause catastrophic result, as described in Chapter 1.
- Likewise, specification mining (Ammons et al., 2002) discovers possible specifications for the (unknown part of the) program by observing its execution and traces, which is also dynamically performed and bears the same problem.
- For static verifiers/analysers, SpaceInvader (Calcagno et al., 2009) simply assumes the program and the unknown procedure have disjoint memory foot-

prints so that the unknown call can be safely ignored due to the hypothetical frame rule (O'Hearn et al., 2004), whereas this assumption does not hold in many cases.

- Some methods (Emami et al., 1994; Gopan and Reps, 2007) try to take into account all possible implementations for the unknown component; however there can be too many such candidates in general, and hence the verification might be infeasible for large-scaled programs.
- Finally, some verifiers will just stop at the first unknown procedure call and provide an incomplete verification (Nguyen et al., 2007), which is obviously undesirable.

Approach and contributions. Compared with the methods stated above, the approach that we propose in this chapter is a different one to the verification of programs that are partially available due to the unknown component/procedure calls. Given a specification $\mathsf{S} = \{\Phi_{pr}\}\ \mathsf{prog}\ \{\Phi_{po}\}$ for the program prog containing calls to an unknown component unknown, our solution is to proceed with the verification for the known fragments of prog, and at the same time infer a specification S_u that is expected for the unknown component unknown based on the calling context(s). The problem of verifying the program prog against the specification S can now be safely reduced to the problem of verifying the component unknown against the inferred specification S_u, provided that the verification of the known fragments does not cause any problems. The inferred specification is subject to a later verification when an implementation or a specification for the unknown component becomes available. This is essentially an improvement of our previous work (Luo et al., 2010a) by extending the program properties to be verified from simple pointer safety to functional correctness of linked data structures. Such properties include structural numerical ones like size and height, relational numerical ones like sortedness, and multiset ones like symbolic content. This chapter makes the following

technical contributions:

- We propose a novel framework in a combined abstract domain (involving both shape and pure properties) for the verification of memory safety and functional correctness of partially available programs with unknown components.
- Our approach is essentially *top-down*, as it can be used to infer the specification for callee procedures based on the specification for the caller procedure. Hence it may benefit the general software development process as a complement for current *bottom-up* approaches (Nguyen et al., 2007; Calcagno et al., 2009).
- We have invented an abduction mechanism which can be applied in this combined domain. It not only can infer shape-based anti-frames for an entailment, but also can discover corresponding pure information (numerical and/or multiset) as well. We also defined a partial order as a guidance for the quality of abduction results.

Outline. Section 6.2 employs a motivating example to informally illustrate our approach. Section 6.3 presents the programming language (catering for unknown calls) for our analysis. Section 6.4 introduces our abductive reasoning. Section 6.5 depicts our verification algorithms, followed by some concluding remarks.

6.2 The Approach

In this section, we illustrate informally, via an example, how our approach verifies a program by inferring the specification for the unknown procedure it invokes.

Example 6.2.1 (Motivating example) *Our goal is to verify the program* `sort` *against the given specification shown in Figure 6.1. According to the specification,*

```
0 Node sort(Node x)
  requires x::llB⟨S⟩
  ensures res::sllB⟨S⟩
1 {
2   if (x == null) return null;
3   else {
4     Node y = unknown(x);
5     Node z = y.next;
6     Node w = sort(z);
7     y.next = w;
8     return y;
9   } }
```

Figure 6.1: A program `sort` calling an unknown procedure `unknown` to be verified.

the procedure takes in a non-empty linked list (`llB`*)* `x` *and returns a sorted list (*`sllB`*) referenced as* `res`*. The (symbolic) content of these two lists are identical (*`S`*). Note that* `sort` *calls an unknown procedure* `unknown` *at line* `4`*. As we do not have available knowledge about it, the discovery of its specifications is essential for both the verification and our understanding of the program (such that we may find out what sorting algorithm this program implements).*

The verification process of the program is illustrated in Figure 6.2 and Figure 6.3. We conduct a forward analysis on the program body starting with the precondition `x`::`llB`⟨`S`⟩ (line `0`). The results of our analysis (e.g. the abstract states) are marked as comments in the code. The analysis carries on until it reaches the unknown procedure call at line `4`.

```
0 Node sort(Node x) requires x::llB⟨S⟩ ensures res::sllB⟨S⟩
1 {  // res is the value returned by the procedure
1a   // Forward analysis begins with current state σ : x::llB⟨S⟩
2   if (x == null) return null;
2a   // σ : x::llB⟨S⟩ ∧ x=null ∧ res=null
2b   // Check whether current state meets the postcondition: σ ⊢ res::sllB⟨S⟩
2c   // It succeeds, and verification on this branch terminates
3   else {
3a   // σ : x::llB⟨S⟩ ∧ x≠null
3b   // Unknown call appears afterwards; extract its precondition from σ
3c   // Φ^u_pr := Local(σ, {x}) := x::llB⟨S⟩ ∧ x≠null
3d   // Also distinguish the frame part not touched by unknown call
3e   // R_0 := Frame(σ, {x}) := emp ∧ x≠null
4   Node y = unknown(x);
4a   // We know nothing about the effect of the unknown call, and thus
4b   // begin to discover its post-effect starting from emp (saved in σ')
4c   // σ'_0 : emp ∧ x=a ∧ y=res_u      σ := R_0 * σ'_0      σ' := σ'_0
```

Figure 6.2: Verification of `sort` calling an unknown procedure `unknown` (part 1).

As afore-shown, the current state before line 4 is $\texttt{x::llB}\langle\texttt{S}\rangle \wedge \texttt{x}\neq\texttt{null}$ (σ at line 3a), from which we generate the precondition for the unknown call. We split σ into two disjoint parts: the local part $\Phi^{\texttt{u}}_{pr}$ (line 3c) that is depended on, and possibly mutated by, the unknown procedure; and the frame part $\texttt{R}_0$ (line 3e) that is not accessed by the unknown procedure. Intuitively, the *local* part of a state w.r.t. a set of variables X is the part of the heap reachable from variables in X (together with pure information); while the *frame* part denotes the unreachable heap part (together with pure information). For example, for a program state $\texttt{x::Node}\langle\texttt{a},\texttt{w}\rangle * \texttt{y::Node}\langle\texttt{b},\texttt{z}\rangle * \texttt{z::Node}\langle\texttt{c},\texttt{null}\rangle \wedge \texttt{w}=\texttt{z}$, its local part w.r.t. {x} is $\texttt{x::Node}\langle\texttt{a},\texttt{w}\rangle * \texttt{z::Node}\langle\texttt{c},\texttt{null}\rangle \wedge \texttt{w}=\texttt{z}$, and its frame part w.r.t. {x} is $\texttt{y::Node}\langle\texttt{b},\texttt{z}\rangle \wedge \texttt{w}=\texttt{z}$. We will have their formal definitions in Section 6.5. Thus we take $\Phi^{\texttt{u}}_{pr}$ (line 3c) as a crude precondition for the unknown procedure, since it denotes the part of program state that is accessible, and hence potentially usable, by the unknown call. The frame part $\texttt{R}_0$ is not touched by the unknown call and will remain in the post-state, as shown in line 4c.

At line 4c, the abstract state after the unknown call (σ) consists of two parts: one is the aforesaid frame $\texttt{R}_0$ not accessed by the call, and the other is the procedure's postcondition which is unfortunately not available. Our next step is to discover the postcondition by examining the code fragment after the unknown call (lines 4d to 8e mainly in Figure 6.3). For this task, a traditional approach might be a backward reasoning from the caller's postcondition towards the unknown call's postcondition. However, this is proven infeasible for separation logic based shape domain by previous works (Calcagno et al., 2009), and hence we employ another approach with a forward reasoning from the unknown call towards the caller's postcondition, using *abduction* to accumulate the unknown call's postcondition.

4d // Next instruction (y.next) requires y be a Node

4e // But entailment checking $\sigma \vdash \texttt{y::Node}\langle \texttt{v}, \texttt{p}\rangle$ fails

4f // Therefore this must be part of the unknown call's post-effect; we use

4g // *abduction* to find it and add it to current state and unknown call's post

4h // $\sigma * [\sigma'_1] \rhd \texttt{y::Node}\langle \texttt{v}, \texttt{p}\rangle$ (s.t. $\sigma * \sigma'_1 \vdash \texttt{y::Node}\langle \texttt{v}, \texttt{p}\rangle * \texttt{true}$)

4i // $\sigma'_1 : \texttt{y::Node}\langle \texttt{v}, \texttt{p}\rangle$ $\quad \sigma := \sigma * \sigma'_1 \quad \sigma' := \sigma' * \sigma'_1$

5 `Node z = y.next;`

5a // Current state $\sigma : \texttt{y::Node}\langle \texttt{v}, \texttt{z}\rangle$

5b // Next instruction invokes this procedure recursively and requires its pre

5c // But $\sigma \vdash \texttt{z::llB}\langle \texttt{S}_1\rangle$ fails again due to lack of knowledge about unknown call

5d // Again we use abduction to find the missing part of unknown call's post-effect

5e // $\sigma * [\sigma'_2] \rhd \texttt{z::llB}\langle \texttt{S}_1\rangle$ (s.t. $\sigma * \sigma'_2 \vdash \texttt{z::llB}\langle \texttt{S}_1\rangle * \texttt{true}$)

5f // $\sigma'_2 : \texttt{z::llB}\langle \texttt{S}_1\rangle$ $\quad \sigma := \sigma * \sigma'_2 \quad \sigma' := \sigma' * \sigma'_2$

6 `Node w = sort(z);`

6a // Current state $\sigma : \texttt{y::Node}\langle \texttt{v}, \texttt{z}\rangle * \texttt{w::sllB}\langle \texttt{S}_1\rangle$ (w already refers to a sorted list)

7 `y.next = w;`

7a // Current state $\sigma : \texttt{y::Node}\langle \texttt{v}, \texttt{w}\rangle * \texttt{w::sllB}\langle \texttt{S}_1\rangle$

8 `return y;`

8a // $\sigma : \texttt{y::Node}\langle \texttt{v}, \texttt{w}\rangle * \texttt{w::sllB}\langle \texttt{S}_1\rangle \wedge \texttt{res}{=}\texttt{y}$; it should imply sort's postcondition

8b // But $\sigma \vdash \texttt{res::sllB}\langle \texttt{S}\rangle$ still fails, suggesting more post-effect of unknown call

8c // A final abduction is conducted to find it: $\sigma * [\sigma'_3] \rhd \texttt{res::sllB}\langle \texttt{S}\rangle$

8d // $\sigma'_3 : \texttt{S}{=}\{\texttt{v}\}{\sqcup}\texttt{S}_1 \wedge \forall \texttt{u}{\in}\texttt{S}_1{\cdot}\texttt{v}{\leq}\texttt{u}$ $\quad \sigma := \sigma * \sigma'_3 \quad \sigma' := \sigma' * \sigma'_3$

8e // All abduction results will be combined at last to form unknown call's post

9 `} }`

9a // $\Phi^{\texttt{u}}_{pr} : \texttt{a::llB}\langle \texttt{S}\rangle \wedge \texttt{a}{\neq}\texttt{null}$ (a is the unknown procedure's formal parameter)

9b // $\Phi^{\texttt{u}}_{po} : \texttt{res}_{\texttt{u}}\texttt{::Node}\langle \texttt{v}, \texttt{b}\rangle * \texttt{b::llB}\langle \texttt{S}_1\rangle \wedge \texttt{S}{=}\{\texttt{v}\}{\sqcup}\texttt{S}_1 \wedge \forall \texttt{u}{\in}\texttt{S}_1{\cdot}\texttt{v}{\leq}\texttt{u}$

Figure 6.3: Verification of sort calling an unknown procedure unknown (part 2).

Initially, we assume the unknown procedure having an empty heap σ'_0 as its post-condition[1], and gradually discover the missing parts of the postcondition during the symbolic execution of the code fragment after the unknown call. To do that, our analysis keeps track of a pair (σ, σ') at each program point, where σ refers to the current heap state, and σ' denotes the expected postcondition discovered so far for the unknown procedure. The notations $\sigma'_{\mathtt{i}}$ are used to represent parts of the discovered postcondition.

At line `5`, `y.next` is dereferenced, whose value is then assigned to `z`. For such dereference to succeed, it requires that `y` be pointing to a node in the heap in the current state. However, we only have an empty heap here (σ in line `4c`). This is not necessarily due to a program error; it might be attributed to the fact that the unknown call's postcondition is still unknown. Therefore, our analysis performs an abduction (line `4h`) to infer the missing part σ'_1 for σ such that $\sigma * \sigma'_1$ implies that `y` points to a `Node`. As shown in line `4i`, σ'_1 is inferred to be $\mathtt{y}{::}\mathtt{Node}\langle \mathtt{v}, \mathtt{p} \rangle$, which is accumulated into σ' as part of the expected postcondition of the unknown procedure. (We will explain the details for abduction in Section 6.4.) Now the heap state combined with the inferred σ'_1 meets the requirement of the dereference, and thus the forward analysis continues.

At line `6`, the procedure `sort` is called recursively. Here the current heap state still does not satisfy the precondition of `sort` (as shown in line `5c`). Blaming the lack of knowledge about the unknown call's postcondition, we conduct another abduction (line `5e`) to infer the missing part σ'_2 for σ such that $\sigma * \sigma'_2$ entails the precondition of `sort` w.r.t. some substitution $[\mathtt{z}/\mathtt{x}]$. Updated with the abduction result $\mathtt{z}{::}\mathtt{llB}\langle \mathtt{S}_1 \rangle$, the program state now meets the precondition of `sort`, which is later transformed

[1]Note that we introduce fresh logical variables `a` and $\mathtt{res}_{\mathtt{u}}$ to record the value of `x` and `y` when `unknown` returns.

to $\texttt{w}{::}\texttt{sllB}\langle \texttt{S}_1 \rangle$ as the effect of sorting over $\texttt{z}$.

After that, line 7 links y and the sorted list w together. Then y is returned as the procedure's result at last. The corresponding state σ at line 8a is expected to establish the postcondition of sort for the overall verification to succeed. However, it does not (as shown in line 8b). Again this might be because part of the unknown call's postcondition is still missing. Therefore, we perform a final abduction (line 8c) to infer the missing σ'_3 as follows:

$$(\texttt{y}{::}\texttt{Node}\langle \texttt{v}, \texttt{w} \rangle * \texttt{w}{::}\texttt{sllB}\langle \texttt{S}_1 \rangle \wedge \texttt{res}{=}\texttt{y}) * [\sigma'_3] \rhd \texttt{res}{::}\texttt{sllB}\langle \texttt{S} \rangle$$

such that $\sigma * \sigma'_3$ implies the postcondition. In this case, our abductor returns σ'_3 as a sophisticated pure constraint $\texttt{S}{=}\{\texttt{v}\} \sqcup \texttt{S}_1 \wedge \forall \texttt{u} {\in} \texttt{S}_1 {\cdot} \texttt{v} {\leq} \texttt{u}$ as the result which is then added into σ', as shown in line 8d.

Finally, we generate the expected pre/post-specification for the unknown procedure (lines 9a and 9b). The precondition is obtained from the local pre-state of the unknown call, $\Phi^{\texttt{u}}_{pr}$ at line 3c, by replacing all variables that are aliases of a with the formal parameter a. The postcondition is obtained from the accumulated abduction result, σ', after performing a similar substitution (which also involves formal parameter $\texttt{res}_\texttt{u}$). Our discovered specification for the unknown procedure Node unknown(Node a) is:

$$\Phi^{\texttt{u}}_{pr} : \texttt{a}{::}\texttt{llB}\langle \texttt{S} \rangle \wedge \texttt{a}{\neq}\texttt{null}$$
$$\Phi^{\texttt{u}}_{po} : \exists \texttt{b} \cdot \texttt{res}_\texttt{u}{::}\texttt{Node}\langle \texttt{v}, \texttt{b} \rangle * \texttt{b}{::}\texttt{llB}\langle \texttt{S}_1 \rangle \wedge \texttt{S}{=}\{\texttt{v}\} \sqcup \texttt{S}_1 \wedge \forall \texttt{u} {\in} \texttt{S}_1 {\cdot} \texttt{v} {\leq} \texttt{u}$$

This derived specification has two implications. The first is that the entire program is verified on the condition that unknown meets such specification. The second is a hint of the behaviours of both the caller (sort) and the callee (unknown), that is, unknown should take as input a list and returns another list with identical content as the input, whose smallest element is exactly at its head. After calling it, sort only needs to sort the rest of the list to accomplish the whole task of sorting. From

this we may guess that `sort` could be a selection-sort algorithm if `unknown` always selects the smallest element and puts it on the list head, or it could be bubble-sort algorithm if `unknown` is a bubbling procedure to exchange two adjacent elements in descending order. Therefore this enhances our understanding of the whole program, and we can verify it as soon as we have the code of `unknown`.

6.3 Programming Language

To accommodate the unknown procedures, we augment the programming language in Figure 3.1 with unknown components as in Figure 6.4. A program *Prog* still consists of two parts: type declarations and procedure declarations. The type declarations remains the same as in Chapter 3. The procedure declarations now include *meth* and *munk*, of which the second contains invocations to unknown procedures while the first does not.

The main part of the language is not altered much: it is still expression-oriented; e is the (recursively defined) program constructor and d and $d[x]$ are atomic instructions; we still allow both call-by-value and call-by-reference method parameters, etc.

To address the unknown calls, we employ *unknown constructors* u and v to denote expressions that involve invocations to the unknown procedures ($unk(\boldsymbol{x}, \boldsymbol{y})$). An *unknown block* v is defined as a sequence of normal expressions sandwiching an *unknown expression* u, which can be a single unknown call, or a sequence of unknown calls, or an if-conditional statement/while loop containing an unknown block. Our aim is to discover the specifications for the unknown procedures in u and v to verify the whole program.

Program	*Prog*	::=	$\boldsymbol{tdecl\ meth\ munk}$
Type declaration	*tdecl*	::=	*classt* | *spred* | *lemma*
Class declaration	*classt*	::=	`class` c { $\boldsymbol{field}$ }
Field declaration	*field*	::=	$t\ x$
Type	t	::=	$c \mid \tau$
Procedure declaration	*meth*	::=	$t\ mn\ ((\boldsymbol{t\ x}); (\boldsymbol{t\ y}))\ \boldsymbol{mspec}\ \{e\}$
Unknown proc. decl.	*munk*	::=	$t\ mn\ ((\boldsymbol{t\ x}); (\boldsymbol{t\ y}))\ mspec\ \{v\}$
Built-in type	τ	::=	`int` | `bool` | `void`
Expression	e	::=	d heap-insensitive atomic
		|	$d[x]$ heap-sensitive atomic
		|	$v{=}e$ assignment
		|	$e_1; e_2$ sequence
		|	$t\ x;\ e$ local variable
		|	`if` $(x)\ e_1$ `else` e_2
		|	`while` $x\ \{e\}$ `where` $\Phi_{pr} \ast\!\!\rightarrow \Phi_{po}$
Heap-insensitive atomic	d	::=	... (Same as in Chapter 3)
Heap-sensitive atomic	$d[x]$	::=	... (Same as in Chapter 3)
Unknown expression	u	::=	$unk(\boldsymbol{x}; \boldsymbol{y})$
		|	$unk(\boldsymbol{x_0}; \boldsymbol{y_0}); e_1; unk_1(\boldsymbol{x_1}; \boldsymbol{y_1}); e_2; \ldots;$ $e_{n-1}; unk_n(\boldsymbol{x_n}; \boldsymbol{y_n})$
		|	`if` $(x)\ v$ `else` e
		|	`if` $(x)\ e$ `else` v
		|	`if` $(x)\ v_1$ `else` v_2
		|	`while` $x\ \{v\}$ `where` $\Phi_{pr} \ast\!\!\rightarrow \Phi_{po}$
Unknown block	v	::=	$e_1; u; e_2$

Figure 6.4: A core (C-like) imperative language.

The operational semantics of the programming language stays the same as in Chapter 3. All the unknown procedure calls are regarded as invocations to procedures whose effect over the pair (s, h) is not known to us. However, to retain the soundness of our verification, we require that such effect conforms to the unknown procedures' specifications once these specifications are discovered.

As we aim to verify both memory safety and functional correctness of programs, our specification language also stays the same as in Chapter 3 in order to express the multiple program properties (shape, numerical and content) and provide support for reasoning over them.

6.4 Enhanced Abduction Mechanism

As shown in Section 6.2, when analysing the code after an unknown call, it is possible that the current state cannot meet the required precondition for the next instruction due to the lack of information about the unknown procedure. Therefore we need to infer the unknown procedure's specification with *abduction* (or abductive reasoning) (Giacobazzi, 1994; Calcagno et al., 2009). It works as follows: for a failed entailment checking $\sigma_1 \vdash \sigma_2 * \texttt{true}$, it attempts to compute an anti-frame σ', such that $\sigma_1 * \sigma' \vdash \sigma_2 * \texttt{true}$ succeeds. For instance, the entailment checking $\texttt{emp} \vdash \texttt{x::llB}\langle\texttt{S}\rangle$ fails as the antecedent contains an empty heap. Then $\texttt{x::llB}\langle\texttt{S}\rangle$ will be found to strengthen the antecedent and validate the entailment $\texttt{emp} * \texttt{x::llB}\langle\texttt{S}\rangle \vdash \texttt{x::llB}\langle\texttt{S}\rangle$. Compared with some previous works of abduction over the shape domain, such as Calcagno et al. (2009) and our pure abduction in Chapter 4, the abduction described here is more enhanced, as it tries to discover both shape and pure information where applicable. Comparatively, the abduction in Calcagno et al. (2009) only works on shape domain with the list segment predicate, and our pure abduction only finds

pure constraints although it operates in a combined shape/pure domain.

An abduction $\sigma_1 * [\sigma'] \rhd \sigma_2$ can also be written as $\sigma_1 * [\sigma'] \rhd \sigma_2 * \sigma_3$, where σ' is the abduction result (the anti-frame), while σ_3 is the frame part obtained with entailment checking $\sigma_1 * \sigma' \vdash \sigma_2 * \texttt{true}$. Its rules are exhibited in Figure 6.5.

$$\frac{\sigma \not\vdash \sigma_1 * \texttt{true} \quad \sigma_1 \vdash \sigma * \sigma' \quad \sigma * \sigma' \vdash \sigma_1 * \sigma_2}{\sigma * [\sigma'] \rhd \sigma_1 * \sigma_2}$$

$$\frac{\begin{array}{c}\sigma \not\vdash \sigma_1 * \texttt{true} \quad \sigma_1 \not\vdash \sigma * \texttt{true} \quad \sigma_0 \in \mathsf{unroll}(\sigma) \quad \mathsf{data\_no}(\sigma_0) \leq \mathsf{data\_no}(\sigma_1) \\ \sigma_0 \vdash \sigma_1 * \sigma' \text{ or } \sigma_0 * [\sigma'_0] \rhd \sigma_1 * \sigma' \quad \sigma''{=}XPure(\sigma') \quad \sigma \wedge \sigma'' \vdash \sigma_1 * \sigma_2\end{array}}{\sigma \wedge [\sigma''] \rhd \sigma_1 * \sigma_2}$$

$$\frac{\begin{array}{c}\sigma \not\vdash \sigma_1 * \texttt{true} \quad \sigma_1 \not\vdash \sigma * \texttt{true} \\ \sigma_1 * [\sigma'_1] \rhd \sigma * \sigma' \quad \sigma''{=}XPure(\sigma') \quad \sigma \wedge \sigma'' \vdash \sigma_1 * \sigma_2\end{array}}{\sigma \wedge [\sigma''] \rhd \sigma_1 * \sigma_2}$$

$$\frac{\sigma \not\vdash \sigma_1 * \texttt{true} \quad \sigma_1 \not\vdash \sigma * \texttt{true} \quad \sigma * \sigma_1 \not\vdash \texttt{false}}{\sigma * [\sigma_1] \rhd \sigma_1 * \sigma_2}$$

Figure 6.5: Abduction rules.

Our abduction deals with four different cases with their corresponding rules. The first rule triggers when the LHS (σ) does not imply the RHS (σ_1) but the RHS implies the LHS with some formula (σ') as the frame. This rule is quite general and applies in many cases, such as the state immediately after an unknown call where we start with `emp` as the heap state. For the example above $\texttt{emp} \not\vdash \texttt{x::llB}\langle\texttt{S}\rangle$, the RHS can entail the LHS with frame $\texttt{x::llB}\langle\texttt{S}\rangle$. The abduction then checks whether σ plus the frame information σ' implies $\sigma_1 * \sigma_2$ for some σ_2 (`emp` in this example), and returns the result $\texttt{x::llB}\langle\texttt{S}\rangle$.

In the case described by the second rule, neither side implies the other, e.g. for

$\texttt{x::sllB}\langle\texttt{S}\rangle$ as LHS (σ) and $\exists \texttt{p},\texttt{u},\texttt{v} \cdot \texttt{x::Node}\langle\texttt{u},\texttt{p}\rangle * \texttt{p::Node}\langle\texttt{v},\texttt{null}\rangle$ as RHS (σ_1). As the shape predicates in the antecedent σ are formed by disjunctions according to their definitions (like sllB), its certain disjunctive branches may imply σ_1. As the rule suggests, to accomplish abduction $\sigma * [\sigma''] \rhd \sigma_1 * \sigma_2$, we first unfold σ ($\sigma_0 \in \mathsf{unroll}(\sigma)$) and try entailment or further abduction with the results (σ_0) against σ_1. If it succeeds with a frame σ', then we first obtain a pure approximation of σ' with *XPure* (Nguyen et al., 2007), and confirm the abduction by ensuring $\sigma \wedge \sigma'' \vdash \sigma_1 * \sigma_2$. For the example above, the abduction returns $|\texttt{S}|{=}2$ as the anti-frame σ' and discovers the nontrivial frame $\texttt{S}{=}\{\texttt{u},\texttt{v}\} \wedge \texttt{u}{\leq}\texttt{v}$ (σ_2). Note the function data_no returns the number of object nodes in a state, e.g. it returns one for $\texttt{x::Node}\langle\texttt{v},\texttt{p}\rangle * \texttt{p::llB}\langle\texttt{T}\rangle$. The unroll unfolds all shape predicates once in σ, normalises the result to a disjunctive form ($\bigvee_{i=1}^{n} \sigma_i$), and returns the result as a set of formulae ($\{\sigma_1,\ldots,\sigma_n\}$). An instance is that it expands $\texttt{x::Node}\langle\texttt{v},\texttt{p}\rangle * \texttt{p::llB}\langle\texttt{T}\rangle$ to be $\{\texttt{x::Node}\langle\texttt{v},\texttt{p}\rangle \wedge \texttt{p}{=}\texttt{null} \wedge \texttt{T}{=}\emptyset, \exists \texttt{u},\texttt{q},\texttt{T}_1 \cdot \texttt{x::Node}\langle\texttt{v},\texttt{p}\rangle * \texttt{p::Node}\langle\texttt{u},\texttt{q}\rangle * \texttt{q::llB}\langle\texttt{T}_1\rangle \wedge \texttt{T}{=}\texttt{T}_1{\cup}\{\texttt{u}\}\}$. The *XPure* is a strengthened version of that in Nguyen et al. (2007), as it also takes pure parts in σ' and keeps them in the resulted pure constraints.

In the third rule, neither side entails the other, and the second rule does not apply, for example $\exists \texttt{p},\texttt{u},\texttt{v} \cdot \texttt{x::Node}\langle\texttt{u},\texttt{p}\rangle * \texttt{p::Node}\langle\texttt{v},\texttt{null}\rangle$ as LHS (σ) and $\exists \texttt{S} \cdot \texttt{x::sllB}\langle\texttt{S}\rangle$ as RHS (σ_1). In this case the antecedent cannot be unfolded as they are already object nodes. As the rule suggests, it reverses two sides of the entailment and applies the second rule to uncover the constraints σ_1' and σ'. Then it checks that the LHS (σ), with σ' added, does imply the RHS (σ_1) before it returns σ'. For the example above, the abduction returns $\texttt{u}{\leq}\texttt{v}$ which is essential for the two nodes to form a sorted list (σ_1).

When an abduction is needed, the first three rules should be tried first; if they do not succeed in finding a solution, then the last rule is invoked to simply add the consequence to the antecedent, provided that they are consistent. It is effective for

situations like $\texttt{x::Node}\langle\_,\_\rangle \not\vdash \texttt{y::Node}\langle\_,\_\rangle$, where we should add $\texttt{y::Node}\langle\_,\_\rangle$ to the LHS directly (since the other three rules do not apply here).

One observation on abduction is that there can be many solutions of the anti-frame σ' for the entailment $\sigma_1 * \sigma' \vdash \sigma_2 * \texttt{true}$ to succeed. For instance, `false` is always a solution but should be avoided where possible. For all possible solutions to an abduction, we can compare their "quality" with a partial order $\preceq$ over SH defined by the entailment relationship ($\vdash$):

$$\sigma_1 \preceq \sigma_2 =_{df} \quad \sigma_2 \vdash \sigma_1 * \texttt{true}$$

and the smaller (weaker) one in two abduction solutions is regarded as better. We prefer to find solutions that are (potentially locally) minimal with respect to $\preceq$ and consistent. However, such solutions are generally not easy to compute and could incur excess cost (with additional disjunction in the analysis). Therefore, our abductive inference is designed more from a practical perspective to discover anti-frames that should be suitable as specifications for unknown procedures, and the partial order $\preceq$ is used to guide the decision choices of our abduction implementation, but not to find the theoretically best solution.

6.5 Verification

This section presents our algorithms to verify programs with unknown calls.

6.5.1 Main Verification Algorithm

Our main verification algorithm is given in Figure 6.6. It verifies an unknown block v (the third parameter) against the given specifications $mspec_v$ (the second parameter).

The first parameter includes the specifications of already available procedures which might be invoked as well as the unknown ones in the program to be verified. Upon successful verification, this algorithm returns specifications that should be met by the unknown procedures in v. If the verification fails, it suggests that the current program cannot meet one or more given specifications due to a potential program bug. The specifications for unknown procedures will be expressed in terms of special variables $\boldsymbol{a}, \boldsymbol{b}$, etc. as in the earlier example.

The algorithm initialises in the first two lines. It distinguishes the body of the unknown block v (as an unknown expression u in between two normal expressions e_1 and e_2), sets up the set $mspec_u$ to store discovered specifications (line 1), and finds the program variables that are potentially accessed by v and u, respectively (prog_var in line 2). Note that $\boldsymbol{x_0}$ and $\boldsymbol{x}$ are the variables read by v and u, and $\boldsymbol{y_0}$ and $\boldsymbol{y}$ are those mutated. For example, if v contains an assignment `y = x` then `x` will be in $\boldsymbol{x_0}$ and `y` in $\boldsymbol{y_0}$.

After the initialisation, for each specification (*requires* Φ_{pr} *ensures* Φ_{po}) to verify against (line 3), the algorithm works in three steps. The first step is to compute the preconditions of u (lines 4–7). It first conducts a symbolic execution from Φ_{pr} over e_1 (the program segment before u) to obtain its post-state, from which the preconditions for u will be extracted (line 4). The symbolic execution is essentially a forward analysis whose details are presented later. If the post-states include `false`, then it means the given Φ_{pr} cannot guarantee e_1's memory safety, and thus fail is returned (line 5). Otherwise, each post-state of e_1 is processed by function Local as a candidate precondition for u (line 7). Intuitively, it extracts the part of each σ reachable from the variables that may be accessed by u, namely, $\boldsymbol{x}$ and $\boldsymbol{y}$. The function Local is defined as follows:

$$\begin{aligned}\mathsf{Local}(\exists \boldsymbol{z} \cdot \kappa \wedge \pi, \{\boldsymbol{x}\}) =_{df}\ & \exists \mathtt{fv}(\sigma) \cup \{\boldsymbol{z}\} \setminus \mathtt{ReachVar}(\kappa \wedge \pi, \{\boldsymbol{x}\}) \cdot \\ & \mathtt{ReachHeap}(\kappa \wedge \pi, \{\boldsymbol{x}\}) \wedge \pi\end{aligned}$$

Algorithm $\mathsf{Verify}(\mathcal{T}, mspec_v, v)$

1 Denote v as $\{\, e_1; u; e_2 \}$; $mspec_u := \emptyset$

2 $(\boldsymbol{x_0}, \boldsymbol{y_0}) := \mathsf{prog\_var}(v)$; $(\boldsymbol{x}, \boldsymbol{y}) := \mathsf{prog\_var}(u)$

3 **foreach** $(requires\ \Phi_{pr}\ ensures\ \Phi_{po}) \in mspec_v$ **do**

4 $\quad \mathsf{S}_0 := [\![e_1]\!]_{\mathcal{T}}\{\, \Phi_{pr} \wedge \boldsymbol{y'_0}{=}\boldsymbol{y_0}\}$

5 $\quad$ **if** $\mathtt{false} \in \mathsf{S}_0$ **then return** fail **endif**

6 $\quad$ **foreach** $\sigma \in \mathsf{S}_0$ **do**

7 $\quad\quad \Phi^u_{pr} := \mathsf{Local}(\sigma, \{\boldsymbol{x}, \boldsymbol{y}\})$

8 $\quad\quad \boldsymbol{z} := \mathsf{fv}(\Phi^u_{pr}) \setminus \{\boldsymbol{x}, \boldsymbol{y}\}$

9 $\quad\quad \mathsf{S} := [\![e_2]\!]^{\mathsf{A}}_{\mathcal{T}}\{([\boldsymbol{b}/\boldsymbol{y}]\,\mathsf{Frame}(\sigma, \{\boldsymbol{x}, \boldsymbol{y}\}) \wedge \boldsymbol{x}{=}\boldsymbol{a} \wedge \boldsymbol{y}{=}\boldsymbol{b} \wedge \boldsymbol{z}{=}\boldsymbol{c}, \mathtt{emp} \wedge \boldsymbol{x}{=}\boldsymbol{a} \wedge \boldsymbol{y}{=}\boldsymbol{b} \wedge \boldsymbol{z}{=}\boldsymbol{c})\}$

10 $\quad\quad \mathsf{S}' := \{\ (\sigma, \sigma') \mid (\sigma, \sigma') {\in} \mathsf{S} \wedge \sigma \vdash \Phi_{po} * \mathtt{true}\ \} \cup$
$\quad\quad\quad\quad \{\ (\sigma * \sigma'', \sigma' * \sigma'') \mid (\sigma, \sigma') {\in} \mathsf{S} \wedge \sigma \nvdash \Phi_{po} {*} \mathtt{true} \wedge \sigma {*} [\sigma''] \rhd \Phi_{po} {*} \mathtt{true}\ \}$

11 $\quad\quad$ **if** $\exists (\sigma, \sigma') {\in} \mathsf{S}'\ .\ \mathsf{fv}(\sigma') \nsubseteq \mathsf{ReachVar}(\sigma, \{\boldsymbol{a}, \boldsymbol{b}\})$ **then return** (fail, σ') **endif**

12 $\quad\quad$ **foreach** $(\sigma, \sigma') \in \mathsf{S}'$ **do**

13 $\quad\quad\quad \Phi^u_{pr} := [\boldsymbol{a}/\boldsymbol{x}, \boldsymbol{b}/\boldsymbol{y}, \boldsymbol{c}/\boldsymbol{z}]\, \Phi^u_{pr}$

14 $\quad\quad\quad \Phi^u_{po} := \mathsf{sub\_alias}(\sigma', \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\})$

15 $\quad\quad\quad g := (\mathsf{fv}(\Phi^u_{pr}) \cap \mathsf{fv}(\Phi^u_{po})) \setminus \{\boldsymbol{a}, \boldsymbol{b}\}$

16 $\quad\quad\quad mspec_u := mspec_u \cup \{(requires\ \exists (\mathsf{fv}(\Phi^u_{pr}) \setminus (g \cup \{\boldsymbol{a}, \boldsymbol{b}\})) \cdot \Phi^u_{pr}\ ensures\ \Phi^u_{po})\}$

17 $\quad\quad$ **end foreach**

18 $\quad$ **end foreach**

19 **end foreach**

20 $\mathcal{T}_u := \mathsf{CaseAnalysis}(\mathcal{T}, mspec_u, u)$

21 **return** $\mathcal{T} \uplus \mathcal{T}_u$

end Algorithm

Figure 6.6: The main verification algorithm.

where $\mathsf{fv}(\sigma)$ stands for all free (program and logical) variables occurring in σ, and $\texttt{ReachVar}(\kappa \wedge \pi, \{\boldsymbol{x}\})$ is the minimal set of variables satisfying the following relation:

$$\{\boldsymbol{x}\} \cup \{z_2 \mid \exists z_1, \pi_1 \cdot z_1 \in \texttt{ReachVar}(\kappa \wedge \pi, \{\boldsymbol{x}\}) \wedge \pi = (z_1 = z_2 \wedge \pi_1)\} \cup \{z_2 \mid \exists z_1, \kappa_1 \cdot z_1 \in \texttt{ReachVar}(\kappa \wedge \pi, v) \wedge \kappa = (z_1 :: c\langle .., z_2, .. \rangle * \kappa_1)\} \subseteq \texttt{ReachVar}(\kappa \wedge \pi, \{\boldsymbol{x}\})$$

That is, it is composed of aliases of $\boldsymbol{x}$ as well as variables reachable from $\boldsymbol{x}$. And the formula $\texttt{ReachHeap}(\kappa \wedge \pi, \{\boldsymbol{x}\})$ denotes the part of κ reachable from $\{\boldsymbol{x}\}$ and is formally defined as the $*$-conjunction of the following set of formulae:

$$\{\kappa_1 \mid \exists z_1, z_2, \kappa_2 \cdot z_1 \in \texttt{ReachVar}(\kappa \wedge \pi, \{\boldsymbol{x}\}) \wedge \kappa = \kappa_1 * \kappa_2 \wedge \kappa_1 = z_1 :: c\langle \ldots, z_2, \ldots \rangle\}$$

The second step is to discover the postconditions for u (lines 9–11). This is mainly completed with another symbolic execution with abduction over e_2 (line 9), whose details are also introduced later. Here we denote u's post-state as $\texttt{emp}$, since its knowledge is not available yet. Therefore, the initial state for the symbolic execution of e_2 is simply the frame part of state not touched by u. The function Frame is formally defined as

$$\mathsf{Frame}(\exists \boldsymbol{z} \cdot \kappa \wedge \pi, \{\boldsymbol{x}\}) =_{df} \exists \boldsymbol{z} \cdot \texttt{UnreachHeap}(\kappa \wedge \pi, \{\boldsymbol{x}\}) \wedge \pi$$

where $\texttt{UnreachHeap}(\exists \boldsymbol{z} \cdot \kappa \wedge \pi, \{\boldsymbol{x}\})$ is the formula consisting of all $*$-conjuncts from κ which are not in $\texttt{ReachHeap}(\exists \boldsymbol{z} \cdot \kappa \wedge \pi, \{\boldsymbol{x}\})$.

The conjunctions $\boldsymbol{x}{=}\boldsymbol{a} \wedge \boldsymbol{y}{=}\boldsymbol{b} \wedge \boldsymbol{z}{=}\boldsymbol{c}$ in line 9 are to keep track of variable snapshot accessed by u using the special variables $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$. Then the symbolic execution returns a set S of pairs (σ, σ') where σ is a possible post-state of e_2 and σ' records the discovered effect of u. However, maybe u still has some effect that is only exposed in the expected postcondition Φ_{po} for the whole program; therefore we need to check whether or not σ can establish Φ_{po}. If not, another abduction $\sigma * [\sigma''] \rhd \Phi_{po}$ is invoked to discover further effect σ'' which is then added into σ'.

There can still be some complication here. Note that the effect discovered during e_2's symbolic execution may not be attributed all over to u; it is also possible that there is a bug in the program, or the given specification is not sufficient. As a consequence of that, the result σ' returned by our abduction may contain more information than what can be expected from u, in which case we cannot simply regard the whole σ' as the postcondition of u. For example, consider the code fragment $\texttt{unk(x); z=y.next}$ with the precondition $\texttt{x::Node}\langle \_, \texttt{null}\rangle$. Before the assignment (and dereference of $\texttt{y.next}$) we use abduction to get $\texttt{y::Node}\langle \_, \_\rangle$. However, noting the fact $\texttt{y}\notin\texttt{ReachVar}(\sigma, \{\texttt{x}\})$ where $\sigma = \texttt{emp}*\texttt{y::Node}\langle \_, \_\rangle$ is the state immediately after the unknown call with the abduction result, we know that from the unknown call's parameters ($\texttt{x}$), $\texttt{y}$ is not reachable, and hence the unknown call will never establish a state to satisfy $\texttt{y::Node}\langle \_, \_\rangle$. In that case we are assured that the program being verified cannot meet its given specification.

To detect such a situation, we introduce the check in line 11. It tests whether the whole abduction result is reachable from variables accessed by u. If not, then the unreachable part cannot be expected from u, which indicates a possible bug in the program or some inconsistency between the program and its given specification. In such cases, the algorithm returns an additional formula that can be used by a further analysis to either identify the bug or strengthen the specification. Recall the example presented in the previous paragraph: since $\texttt{y::Node}\langle \_, \_\rangle$ cannot be established by the unknown call, if we add it to the precondition of the code fragment (to form a new precondition $\texttt{x::Node}\langle \_, \texttt{null}\rangle * \texttt{y::Node}\langle \_, \_\rangle$), then the verification with the new specification can move on and will potentially succeed.

The third step (lines 12–17) is to form the derived specifications for u in terms of variables $\boldsymbol{a}, \boldsymbol{b}$ and g. Here g denotes logical variables not explicitly accessed by u, but occurring in both pre- and postconditions (ghost variables). The formula $\mathsf{sub\_alias}(\sigma', \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\})$ is obtained from σ' by replacing all variables with their aliases

in $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\}$. It is defined as

$$\begin{aligned}\mathsf{sub\_alias}(\sigma', \{\boldsymbol{x}\}) =_{df} \quad & (\{[x/x_0] \mid x{\in}\{\boldsymbol{x}\} \wedge x_0{\in}\mathsf{aliases}(\sigma', x)\}\ \sigma') \wedge \\ & \bigwedge \{x = x_0 \mid x, x_0{\in}\{\boldsymbol{x}\} \wedge x_0{\in}\mathsf{aliases}(\sigma', x)\}\end{aligned}$$

where a set of substitutions before a formula σ' denotes the result of applying each of those substitutions to σ' (where the ordering is not important), and $\mathsf{aliases}(\sigma', x)$ returns all the aliases of x in σ'.

Finally, at line 20, the obtained specifications $mspec_u$ for u are passed to the case analysis algorithm (given in Figure 6.7) to derive the specifications of unknown procedures invoked in u.

6.5.2 Case Analysis Algorithm

In order to discover specifications for unknown procedures invoked in u, the algorithm in Figure 6.7 conducts a case analysis according to the structure of u. In the first case (line 2), u is simply a single unknown call. In this situation, the algorithm returns all the pre-/postcondition pairs from $mspec_u$ as the unknown procedure's specifications.

In the second case (line 4), u is an `if`-conditional and both branches contain an unknown block. The algorithm uses the main algorithm to verify the two branches separately with preconditions $\Phi_{pr} \wedge x$ and $\Phi_{pr} \wedge \neg x$ respectively, where Φ_{pr} is one of the preconditions of the whole `if`. The results obtained from the two branches are then combined using the $\uplus$ operator:

$$R_1 \uplus R_2 =_{df} \{(\mathsf{f}, \mathsf{Refine}(mspec^1_\mathsf{f} \cup mspec^2_\mathsf{f})) \mid (\mathsf{f}, mspec^1_\mathsf{f}){\in}R_1 \wedge (\mathsf{f}, mspec^2_\mathsf{f}){\in}R_2\}$$

where Refine is used to eliminate any specification ($requires\ \Phi_{pr}\ ensures\ \Phi_{po}$) from a set if there exists a "stronger" one ($requires\ \Phi'_{pr}\ ensures\ \Phi'_{po}$) such that $\Phi'_{pr} \preceq \Phi_{pr}$

Algorithm $\mathsf{CaseAnalysis}(\mathcal{T}, mspec_u, u)$

1 **switch** u

2 **case** $unk(\boldsymbol{x}; \boldsymbol{y})$

3 **return** $\{ (unk(\boldsymbol{x}; \boldsymbol{y}), mspec_u)\}$

4 **case** `if` (x) v_1 `else` v_2

5 $mspec_T := \{(requires\ \Phi_{pr} \wedge x\ ensures\ \Phi_{po}) \mid$
$(requires\ \Phi_{pr}\ ensures\ \Phi_{po}) \in mspec_u\}$

6 $mspec_F := \{(requires\ \Phi_{pr} \wedge \neg x\ ensures\ \Phi_{po}) \mid$
$(requires\ \Phi_{pr}\ ensures\ \Phi_{po}) \in mspec_u\}$

7 $R_1 := \mathsf{Verify}(\mathcal{T}, mspec_T, v_1)$

8 $R_2 := \mathsf{Verify}(\mathcal{T}, mspec_F, v_2)$

9 **return** $R_1 \uplus R_2$

10 **case** `if` (x) v `else` e

11 $mspec_T := \{(requires\ \Phi_{pr} \wedge x\ ensures\ \Phi_{po}) \mid$
$(requires\ \Phi_{pr}\ ensures\ \Phi_{po}) \in mspec_u\}$

12 $R := \mathsf{Verify}(\mathcal{T}, mspec_T, v)$

13 **if** $\exists (requires\ \Phi_{pr}\ ensures\ \Phi_{po}) \in mspec_u, \sigma \in [\![e]\!]_{\mathcal{T}}\{\Phi_{pr} \wedge \neg x\} \cdot$
σ=`false` $\vee\ \sigma \not\models \Phi_{po} * $`true` **then return** fail

14 **else return** R **endif**

15 **case** `if` (x) e `else` v (Similar to the previous case)

16 **case** `while` x $\{ v\}$ `where` $\Delta_{pr} \mathrel{*\!\!\rightarrow} \Delta_{po}$

17 **return** $\mathsf{Verify}(\mathcal{T}, requires\ \Delta_{pr}\ ensures\ \Delta_{po}, v)$

18 **case** $unk_0(\boldsymbol{x_0}; \boldsymbol{y_0})\ \{\ ; e_i; unk_i(\boldsymbol{x_i}; \boldsymbol{y_i})\}_{i=1}^{n}$

19 **return** $\{ (unk_i(\boldsymbol{x_i}; \boldsymbol{y_i}), \mathsf{SeqUnkCalls}(\mathcal{T}, mspec_u, u))\}_{i=0}^{n}$

end Algorithm

Figure 6.7: The case analysis algorithm.

and $\Phi_{po} \preceq \Phi'_{po}$. It is defined as

$$\mathsf{Refine}(\emptyset) =_{df} \emptyset$$
$$\mathsf{Refine}(\{(\mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po})\} \cup \mathsf{Spec}) =_{df}$$
$$\quad \textbf{if } \exists(\mathit{requires}\ \Phi'_{pr}\ \mathit{ensures}\ \Phi'_{po}) \in \mathsf{Spec} \cdot \Phi'_{pr} \preceq \Phi_{pr} \wedge \Phi_{po} \preceq \Phi'_{po}$$
$$\quad \textbf{then } \mathsf{Refine}(\mathsf{Spec}) \textbf{ else } \{(\mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po})\} \cup \mathsf{Refine}(\mathsf{Spec})$$

and $\uplus$ is to refine the union of two specification sets.

The third and fourth cases (lines 10 and 15) are for `if`-conditionals which contain only one unknown block in one of the two branches. This is handled in a similar way as in the second case. The only difference is, for the branch without unknown blocks, we need to verify it with the underlying semantics (line 13).

The fifth case is the `while` loop. As we assume that its specification $(\Delta_{pr} \ast\!\!\rightarrow \Delta_{po})$ is already given for verification in mspec_u, we simply verify its body with the main algorithm (line 17).

In the last case (line 21), where u consists of multiple unknown procedure calls in sequence, another algorithm SeqUnkCalls is invoked to deal with it.

6.5.3 Verifying Sequential Unknown Calls

We provide a solution to the most complicated case (unknown procedure calls in sequence) under a strong assumption, namely, we can find a common specification to capture all these unknown procedures' behaviours. First we illustrate the brief idea using two sequential unknown procedure calls as an example, followed by the general algorithm.

Suppose we have

$$\{\Phi_{pr}\}\ \{unk_0(\boldsymbol{x_0};\boldsymbol{y_0}); e; unk_1(\boldsymbol{x_1};\boldsymbol{y_1})\}\ \{\Phi_{po}\}$$

where e is the only known code fragment within the block. The algorithm works in three steps. In the first step, it extracts the precondition for the first procedure, say Φ^u_{pr}, from the given precondition Φ_{pr} by extracting the part of heap that may be accessed by the call via $\boldsymbol{x_0}$ and $\boldsymbol{y_0}$, which is similar to the first step of the main algorithm Verify. Aiming at a general specification for both unknown calls, it then assumes that the second procedure has a similar precondition Φ^u_{pr}. In the second step, it symbolically executes the code fragment e with the help of the abductor, to discover a crude postcondition, say Φ^0_{po}, expected from the first unknown call. This is similar to the second step of the main algorithm Verify, except that the postcondition for e is now assumed to be Φ^u_{pr}. In the third step, the algorithm takes Φ^0_{po} (with appropriate variable substitutions) as the postcondition of the second unknown call, and checks whether or not the derived post (Φ^0_{po}) satisfies Φ_{po}. If not, it invokes another abduction to strengthen Φ^0_{po} to obtain the final postcondition Φ^u_{po} for the unknown procedures. Note that this strengthening does not affect soundness: the strengthened Φ^u_{po} can still be used as a general postcondition for both unknown procedures.

Figure 6.8 and Figure 6.9 present the algorithm to infer specifications for n ($n \geq 2$) unknown calls in sequence.

As aforementioned, given a block of (n+1) unknown procedure calls with n pieces of known code blocks sandwiched among them ($unk_0(\boldsymbol{x_0};\boldsymbol{y_0})$ $\{; e_i; unk_i(\boldsymbol{x_i};\boldsymbol{y_i})\}_{i=1}^{n}$ in line 1), and the specification (*requires* Φ_{pr} *ensures* Φ_{po}) (line 3) for such a block, our approach generally works in three steps: first, to compute a precondition for the unknown calls; second, to verify each code fragment e_i ($i = 1, \ldots, n$) with abduction to collect expected behaviour of the unknown calls (as part of their postcondition);

Algorithm $\mathsf{SeqUnkCalls}(\mathcal{T}, mspec_u, u)$

1 Denote u as $unk_0(\boldsymbol{x_0}; \boldsymbol{y_0})\ \{; e_i; unk_i(\boldsymbol{x_i}; \boldsymbol{y_i})\}_{i=1}^{n}$

2 $R := \emptyset$

3 **foreach** $(requries\ \Phi_{pr}\ ensures\ \Phi_{po}) \in mspec_u$ **do**

4 $\quad \Phi^u_{pr} := \mathsf{Local}(\Phi_{pr}, \{\boldsymbol{x_0}, \boldsymbol{y_0}\})$

5 $\quad \boldsymbol{z_0} := \mathsf{fv}(\Phi^u_{pr}) \setminus \{\boldsymbol{x_0}, \boldsymbol{y_0}\}$

6 $\quad \Phi^u_{pr} := [\boldsymbol{a}/\boldsymbol{x_0}, \boldsymbol{b}/\boldsymbol{y_0}, \boldsymbol{c}/\boldsymbol{z_0}]\Phi^u_{pr}$

7 $\quad \mathsf{S}'_0 := \{(\Phi_{pr} \wedge \boldsymbol{y_0}'{=}\boldsymbol{y_0}, \mathtt{emp} \wedge \boldsymbol{a}{=}\boldsymbol{x_0} \wedge \boldsymbol{b}{=}\boldsymbol{y_0} \wedge \boldsymbol{c}{=}\boldsymbol{z_0})\}$

8 $\quad$ **for** $i := 1$ **to** n **do**

9 $\quad\quad \mathsf{S}_i := [\![e_i]\!]^{\mathsf{A}}_{\mathcal{T}}\{\ (\Phi^{i-1}_{po} * [\boldsymbol{b}/\boldsymbol{y}_{i-1}]\ \mathsf{Frame}(\sigma_{i-1}, \{\boldsymbol{x}_{i-1}, \boldsymbol{y}_{i-1}\}), \Phi^{i-1}_{po})\ \mid$
$(\sigma_{i-1}, \sigma'_{i-1}){\in}\mathsf{S}'_{i-1} \wedge \Phi^{i-1}_{po} = ([\boldsymbol{x}_{i-1}/\boldsymbol{a}, \boldsymbol{y}_{i-1}/\boldsymbol{b}, \boldsymbol{z}_{i-1}/\boldsymbol{c}]\,\mathsf{sub\_alias}($
$\sigma'_{i-1}, \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\})) \wedge \boldsymbol{a}{=}\boldsymbol{x}_{i-1} \wedge \boldsymbol{b}{=}\boldsymbol{y}_{i-1} \wedge \boldsymbol{c}{=}\boldsymbol{z}_{i-1}\}$ where $\boldsymbol{z}_{i-1}$ is fresh

10 $\quad\quad \mathsf{S}'_i := \{(\sigma, \sigma') \mid (\sigma, \sigma'){\in}\mathsf{S}_i \wedge \rho\sigma \vdash \exists \boldsymbol{c} \cdot \Phi^u_{pr} {*} \mathtt{true}\} \cup \{(\sigma * \sigma'', \sigma' * \sigma'') \mid$
$(\sigma, \sigma'){\in}\mathsf{S}_i \wedge \rho\sigma \nvdash \exists \boldsymbol{c} \cdot \Phi^u_{pr} {*} \mathtt{true} \wedge \rho\sigma{*}[\sigma''] \rhd \exists \boldsymbol{c} \cdot \Phi^u_{pr}\}$
where $\rho{=}[\boldsymbol{a}/\boldsymbol{x_i}, \boldsymbol{b}/\boldsymbol{y_i}]$

11 $\quad\quad$ **if** $\exists(\sigma, \sigma'){\in}\mathsf{S}'_i \cdot \mathsf{fv}(\sigma') \not\subseteq \mathsf{ReachVar}(\sigma, \{\boldsymbol{a}, \boldsymbol{b}\})$ **then return** (fail, σ') **endif**

12 $\quad$ **end for**

Figure 6.8: Algorithm for sequential unknown calls (part 1).

13 $\mathsf{S}_{n+1} := \{\ (\Phi^n_{po} * [\boldsymbol{b}/\boldsymbol{y_n}]\mathsf{Frame}(\sigma_n, \{\boldsymbol{x_n}, \boldsymbol{y_n}\}), \Phi^n_{po}) \mid (\sigma_n, \sigma'_n) \in \mathsf{S}'_n \wedge$
$\Phi^n_{po} = ([\boldsymbol{x_n}/\boldsymbol{a}, \boldsymbol{y_n}/\boldsymbol{b}, \boldsymbol{z_n}/\boldsymbol{c}]\,\mathsf{sub\_alias}(\sigma'_n, \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\})) \wedge$
$\boldsymbol{a}{=}\boldsymbol{x_n} \wedge \boldsymbol{b}{=}\boldsymbol{y_n} \wedge \boldsymbol{c}{=}\boldsymbol{z_n}\}$ where $\boldsymbol{z_n}$ is fresh

14 $\mathsf{S}'_{n+1} := \{(\sigma, \sigma') \mid (\sigma, \sigma') \in \mathsf{S}_{n+1} \wedge \sigma \vdash \Phi_{po} * \mathtt{true}\} \cup \{(\sigma * \sigma'', \sigma' * \sigma'') \mid$
$(\sigma, \sigma') \in \mathsf{S}_{n+1} \wedge \sigma \not\vdash \Phi_{po} * \mathtt{true} \wedge \sigma * [\sigma''] \rhd \Phi_{po}\}$

15 **if** $\exists (\sigma, \sigma') \in \mathsf{S}'_{n+1} \cdot \mathsf{fv}(\sigma') \not\subseteq \mathsf{ReachVar}(\sigma, \{\boldsymbol{a}, \boldsymbol{b}\})$ **then return** (fail, σ') **endif**

16 **foreach** $(\sigma, \sigma') \in \mathsf{S}'_{n+1}$ **do**

17 $\Phi^u_{po} := \mathsf{sub\_alias}(\sigma', \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\})$

18 $g := \mathsf{fv}(\Phi^u_{pr}) \cap \mathsf{fv}(\Phi^u_{po}) \setminus \{\boldsymbol{a}, \boldsymbol{b}\}$

19 $R := \mathsf{Refine}(R \cup \{(\mathit{requires}\ \exists \mathsf{fv}(\Phi^u_{pr}) \setminus (g \cup \{\boldsymbol{a}, \boldsymbol{b}\}) \cdot \Phi^u_{pr}\ \mathit{ensures}\ \Phi^u_{po})\})$

20 **end foreach**

21 **end foreach**

22 **return** R

end Algorithm

Figure 6.9: Algorithm for sequential unknown calls (part 2).

third, to guarantee that the collected postcondition satisfies Φ_{po}. If not, then another abduction is conducted to strengthen the gained postcondition to ensure this.

The first step is completed by lines 4 to 6. The local part of Φ_{pr} is extracted w.r.t. the first unknown call's parameters $\boldsymbol{x_0}$ and $\boldsymbol{y_0}$. Other free variables are distinguished as $\boldsymbol{z_0}$, which may be ghost variables. Finally the precondition is found in terms of special logical variables $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$.

The second step is performed over each $e_i; unk_i(\boldsymbol{x_i}; \boldsymbol{y_i})$. Its main idea is to take the postcondition generated for the last unknown call (Φ_{po}^{i-1}), plus the frame part during the entailment check against Φ_{pr}^{i-1}, as the post-state of $unk_{i-1}(\boldsymbol{x_{i-1}}; \boldsymbol{y_{i-1}})$, and try to verify e_i beginning with such a state, using abduction when necessary (line 9). After the verification we get S_i containing abstract states before $unk_i(\boldsymbol{x_i}; \boldsymbol{y_i})$, and we want those states to satisfy its precondition Φ_{pr}^{u} subject to substitution. Note that during the verification of e_i and the last satisfaction checking we may use abduction to strengthen the program state, whose results reflect the expected behaviour of $unk_{i-1}(\boldsymbol{x_{i-1}};\ \boldsymbol{y_{i-1}})$ and are accumulated as its expected postcondition. Hence we achieve a sufficiently strong postcondition for each unknown call.

The third step is similar to the first algorithm: it checks whether the final abstract state entails the postcondition of the whole block, and strengthens the final abstract state with abduction if it cannot. Then the ghost variables are recognised and processed analogously to the first algorithm. Finally the strongest specifications discovered for those unknown procedures are returned.

Note that our current solution tries to find a common specification (*requires* Φ_{pr} *ensures* Φ_{po}) suitable for all the unknown procedures. Generally we may allow the unknown procedures to have different specifications. In theory, this can be achieved by a more in-depth analysis which examines the known code fragments in between

those unknown calls. That is, by analysing the code fragment e_i we would hopefully obtain a postcondition for the $(i-1)$-th procedure and a precondition for the i-th. In the case of two unknown calls $unk_0(\boldsymbol{x_0}; \boldsymbol{y_0}); e_1; unk_1(\boldsymbol{x_1}; \boldsymbol{y_1})$, the precondition for unk_0 and the postcondition for unk_1 can be obtained as usual (by analysing the code before unk_0 and after unk_1 respectively). To derive the postcondition Φ^0_{po} for unk_0 and the precondition Φ^1_{pr} for unk_1, we initialise Φ^0_{po} to be `emp` to start a forward analysis over e_1 with abduction, to accumulate (via abduction) the expected behaviour of unk_0 (for e_1 to be verified) as Φ^0_{po}, and extract a formula (which is relevant to the footprint of unk_1) from the abstract state at the end of e_1 as Φ^1_{pr}. However, our initial experiments show that, unless the fragment e_1 is sufficiently complex to expose enough information expected from unk_0, the derived Φ^0_{po} and Φ^1_{pr} can be rather weak. As a consequence, the derived specification for unk_0 can be too weak (with a weak postcondition) and the one for unk_1 can be too strong (with a weak precondition). It remains an open problem how we might tune the derived results to obtain more reasonable specifications. We conjecture that certain heuristics might help and we will explore this further in our future work.

6.5.4 Abstract Semantics

As shown in the algorithms, we use two kinds of abstract semantics to analyse the program: an underlying semantics and another semantics based on both the first one and abduction to improve the postcondition of unknown calls.

Our underlying abstract semantics is generally similar as the one in Chapter 4 yet with some subtle differences (such as procedure invocation). Its type is defined as

$$[\![e]\!] \ : \ \mathsf{AllSpec} \rightarrow \mathcal{P}_{\mathsf{SH}} \rightarrow \mathcal{P}_{\mathsf{SH}}$$

where $\mathsf{AllSpec}$ contains procedure specifications (extracted from the program *Prog*).

For some expression e, given its precondition, the semantics will calculate the post-condition.

The foundation of the semantics is the basic transition functions from a conjunctive abstract state (σ) to a conjunctive or disjunctive abstract state (σ or Δ) below:

$$\begin{array}{lcll} \mathsf{unfold}(x) & : & \mathsf{SH} \rightarrow \mathcal{P}_{\mathsf{SH}[x]} & \text{Unfolding} \\ \mathsf{exec}(d[x]) & : & \mathsf{AllSpec} \rightarrow \mathsf{SH}[x] \rightarrow \mathsf{SH} & \text{Heap-sensitive execution} \\ \mathsf{exec}(d) & : & \mathsf{AllSpec} \rightarrow \mathsf{SH} \rightarrow \mathsf{SH} & \text{Heap-insensitive execution} \end{array}$$

where $\mathsf{SH}[x]$ denotes the set of conjunctive abstract states in which each element has x exposed as the head of an object node ($x{::}c\langle \boldsymbol{v} \rangle$), and $\mathcal{P}_{\mathsf{SH}[x]}$ contains all the (disjunctive) abstract states, each of which is composed by such conjunctive states. Here $\mathsf{unfold}(x)$ rearranges the symbolic heap so that the cell referred to by x is exposed for access by heap sensitive commands $d[x]$ via the second transition function $\mathsf{exec}(d[x])$. The third function defined for other (heap insensitive) commands d does not require such exposure of x.

The unfolding function is defined by the following two rules:

$$\frac{\mathit{isobj}(c) \quad \sigma \vdash x{::}c\langle \boldsymbol{v} \rangle * \sigma'}{\mathsf{unfold}(x)\sigma \rightsquigarrow \sigma} \qquad \frac{\mathit{isspred}(c) \quad \sigma \vdash x{::}c\langle \boldsymbol{u} \rangle * \sigma' \quad \mathtt{root}{::}c\langle \boldsymbol{v} \rangle \equiv \Phi}{\mathsf{unfold}(x)\sigma \rightsquigarrow \sigma' * [x/\mathtt{root}, \boldsymbol{u}/\boldsymbol{v}]\Phi}$$

The test $\mathit{isobj}(c)$ returns `true` only if c is an object node and $\mathit{isspred}(c)$ returns `true` only if c is a shape predicate.

The symbolic execution of heap-sensitive commands $d[x]$ (i.e. $x.f_i$, $x.f_i := w$, or $\mathtt{free}(x)$) assumes that the rearrangement $\mathsf{unfold}(x)$ has been done prior to the exe-

cution:

$$\frac{\mathit{isobj}(c) \qquad \sigma \vdash x{::}c\langle v_1, \ldots, v_n\rangle * \sigma'}{\mathsf{exec}(x.f_i)(\mathcal{T})\sigma \rightsquigarrow \sigma' * x{::}c\langle v_1, \ldots, v_n\rangle \wedge \mathtt{res}{=}v_i}$$

$$\frac{\mathit{isobj}(c) \qquad \sigma \vdash x{::}c\langle v_1, \ldots, v_n\rangle * \sigma'}{\mathsf{exec}(x.f_i := w)(\mathcal{T})\sigma \rightsquigarrow \sigma' * x{::}c\langle v_1, \ldots, v_{i-1}, w, v_{i+1}, \ldots, v_n\rangle}$$

$$\frac{\mathit{isobj}(c) \qquad \sigma \vdash x{::}c\langle \boldsymbol{u}\rangle * \sigma'}{\mathsf{exec}(\mathtt{free}(x))(\mathcal{T})\sigma \rightsquigarrow \sigma'}$$

The symbolic execution rules for heap-insensitive commands are as follows:

$$\mathsf{exec}(k)(\mathcal{T})\sigma \rightsquigarrow \sigma \wedge \mathtt{res}{=}k \qquad\qquad \mathsf{exec}(x)(\mathcal{T})\sigma \rightsquigarrow \sigma \wedge \mathtt{res}{=}x$$

$$\frac{\mathit{isobj}(c)}{\mathsf{exec}(\mathtt{new}\ c(\boldsymbol{v}))(\mathcal{T})\sigma \rightsquigarrow \sigma * \mathtt{res}{::}c\langle \boldsymbol{v}\rangle}$$

$$\frac{\begin{array}{c} t\ mn\ ((t_i\ u_i)_{i=1}^m; (t_i\ v_i)_{i=1}^n)\ \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \in \mathcal{T} \\ \rho = [x'_i/u_i]_{i=1}^m \circ [y'_i/v_i]_{i=1}^n \qquad \sigma \vdash \rho\Phi_{pr} * \sigma' \\ \rho_o = [r_i/v_i]_{i=1}^n \circ [x'_i/u'_i]_{i=1}^m \circ [y'_i/v'_i]_{i=1}^n \quad \rho_l = [r_i/y'_i]_{i=1}^n \quad \mathit{fresh\ logical}\ r_i \end{array}}{\mathsf{exec}(mn(x_1, \ldots, x_m; y_1, \ldots, y_n))(\mathcal{T})\sigma \rightsquigarrow (\rho_l\sigma') * (\rho_o\Phi_{po})}$$

Note that the first three rules deal with constant (k), variable (x) and object node creation ($\mathtt{new}\ c(\boldsymbol{v})$), respectively, while the last rule handles method invocation. In the last rule, the call site is ensured to meet the precondition of mn, as signified by $\sigma \vdash \rho\Phi_{pr} * \sigma'$. In this case, the execution succeeds and the post-state of the method call involves mn's postcondition as signified by $\rho_o\Phi_{po}$.

A lifting function † is defined to lift unfold's domain to $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{unfold}^{\dagger}(x) \bigvee \sigma_i =_{df} \bigvee(\mathsf{unfold}(x)\sigma_i)$$

and this function is overloaded for exec to lift both its domain and range to $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{exec}^{\dagger}(d)(\mathcal{T}) \bigvee \sigma_i =_{df} \bigvee(\mathsf{exec}(d)(\mathcal{T})\sigma_i)$$

Based on the transition functions above, we can define the abstract semantics for a

program expression e as follows:

$$
\begin{array}{lll}
[\![d[x]]\!]_{\mathcal{T}}\Delta & =_{df} & \mathsf{exec}^{\dagger}(d[x])(\mathcal{T}) \circ \mathsf{unfold}^{\dagger}(x)\Delta \\
[\![d]\!]_{\mathcal{T}}\Delta & =_{df} & \mathsf{exec}^{\dagger}(d)(\mathcal{T})\Delta \\
[\![e_1; e_2]\!]_{\mathcal{T}}\Delta & =_{df} & [\![e_2]\!]_{\mathcal{T}} \circ [\![e_1]\!]_{\mathcal{T}}\Delta \\
[\![x := e]\!]_{\mathcal{T}}\Delta & =_{df} & [x_1/x', r_1/\mathtt{res}]([\![e]\!]_{\mathcal{T}}\Delta) \wedge x'{=}r_1 \quad \textit{fresh logical } x_1, r_1 \\
[\![\mathtt{if}\ (v)\ e_1\ \mathtt{else}\ e_2]\!]_{\mathcal{T}}\Delta & =_{df} & ([\![e_1]\!]_{\mathcal{T}}(v{\wedge}\Delta)) \vee ([\![e_2]\!]_{\mathcal{T}}(\neg v{\wedge}\Delta))
\end{array}
$$

$$
\frac{\Delta \vdash \Phi_{pr} * \mathtt{R}}{[\![\mathtt{while}\ x\ \{e\}\ \mathtt{where}\ \Phi_{pr} \mathrel{*\!\!\rightarrow} \Phi_{po}]\!]_{\mathcal{T}}\Delta =_{df} \mathtt{R} * \Phi_{po} \wedge \neg x}
$$

Next we define the abstract semantics with abduction used in our analysis, whose type is

$$[\![e]\!]^{\mathsf{A}} \ : \ \mathsf{AllSpec} \to \mathcal{P}(\mathsf{SH} \times \mathsf{SH}) \to \mathcal{P}(\mathsf{SH} \times \mathsf{SH})$$

It takes a piece of program and a specification table, to map a (disjunctive) set of pair of symbolic heaps to another such set (where the first in the pair is the current state and the second is the abduction result).

This semantics also consists of the basic transition functions which compose the atomic instructions' semantics and then the program constructors' semantics. Here the basic transition functions are lifted as

$$
\begin{array}{l}
\mathsf{Unfold}(x)(\sigma, \sigma') =_{df} \\
\quad \textbf{let}\ \Delta{=}\mathsf{unfold}(x)\sigma \text{ and } \mathsf{S}{=}\{(\sigma_1, \sigma') \mid \sigma_1 \in \Delta\} \\
\quad \textbf{in if}\ (\mathtt{false} \notin \Delta)\ \textbf{then}\ \mathsf{S} \\
\qquad \textbf{else if}\ (\Delta \vdash x{=}a \text{ for some } a{\in}\mathsf{SVar}) \text{ and} \\
\qquad\qquad (\sigma' \not\vdash a{::}c\langle \boldsymbol{y} \rangle * \mathtt{true} \text{ for fresh } \{\boldsymbol{y}\}{\subseteq}\mathsf{LVar}) \\
\qquad \textbf{then}\ \mathsf{S} \cup \{(\sigma_1 * x{::}c\langle \boldsymbol{y} \rangle, \sigma' * x{::}c\langle \boldsymbol{y} \rangle) \mid \sigma_1 \in \Delta\} \\
\qquad \textbf{else}\ \ \mathsf{S} \cup \{(\mathtt{false}, \sigma')\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Exec}(ds)(\sigma, \sigma') =_{df} \textbf{let}\ \sigma_1{=}\mathsf{exec}(ds)\sigma\ \textbf{in}\ \{(\sigma_1, \sigma')\} \\
\quad \text{where } ds \text{ is either } d[x] \text{ or } d, \text{ except procedure call}
\end{array}
$$

In the definition of Unfold we view Δ as a disjunctive set of conjunctive formulae, and mean by $\sigma \in \Delta$ that σ is one branch of Δ. Meanwhile SVar is a set of special logical variables used to record the program's footprint. In the definition of Exec we need special treatment for instructions that may alter variable values, say procedure call. As can be seen in the rule below, when a call-by-reference variable y is assigned to a new value after the call, the original value is still preserved with a substitution $\rho = [y_0/y]$ where y_0 is fresh. Doing this allows us to keep the connection among the history values of a variable and its latest value, which may be essential as a link from the unknown procedure's postcondition to its caller's postcondition.

$$\frac{\begin{array}{c} t\ mn\ ((t_i\ u_i)_{i=1}^m; (t_i\ v_i)_{i=1}^n)\ \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \in \mathcal{T} \\ \rho = [x_i'/u_i]_{i=1}^m \circ [y_i'/v_i]_{i=1}^n \qquad \sigma \vdash \rho\Phi_{pr} * \sigma_1 \text{ and } \sigma_1'{=}\mathtt{emp},\ \text{or } \sigma * [\sigma_1'] \rhd \rho\Phi_{pr} * \sigma_1 \\ \rho_o = [r_i/v_i]_{i=1}^n \circ [x_i'/u_i']_{i=1}^m \circ [y_i'/v_i']_{i=1}^n \quad \rho_l = [r_i/y_i']_{i=1}^n \quad \mathit{fresh\ logical}\ r_i \end{array}}{\mathsf{Exec}(mn(x_1,..,x_m;y_1,..,y_n))(\mathcal{T})(\sigma,\sigma') \rightsquigarrow \{(\sigma_2, \rho_o(\sigma' * \sigma_1')) \mid \sigma_2 \in (\rho_l\sigma_1) * (\rho_o\Phi_{po})\}}$$

A similar lifting function † is defined to lift Unfold's and Exec's domains:

$$\begin{array}{lll} \mathsf{Unfold}^\dagger(x)\{(\sigma_i,\sigma_i')\} & =_{df} & \bigcup(\mathsf{Unfold}(x)(\sigma_i,\sigma_i')) \\ \mathsf{Exec}^\dagger(ds)(\mathcal{T})\{(\sigma_i,\sigma_i')\} & =_{df} & \bigcup(\mathsf{Exec}(ds)(\mathcal{T})(\sigma_i,\sigma_i')) \end{array}$$

Based on the above transition functions, the abstract semantics with abduction is as follows:

$$\begin{array}{lll} [\![d[x]]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} & =_{df} & \mathsf{Exec}^\dagger(d[x])(\mathcal{T}) \circ \mathsf{Unfold}^\dagger(x)\{(\sigma,\sigma')\} \\ [\![d]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} & =_{df} & \mathsf{Exec}^\dagger(d)(\mathcal{T})\{(\sigma,\sigma')\} \\ [\![e_1;e_2]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} & =_{df} & [\![e_2]\!]^{\mathsf{A}}_{\mathcal{T}} \circ [\![e_1]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} \\ [\![\mathtt{if}\ (v)\ e_1\ \mathtt{else}\ e_2]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} & =_{df} & ([\![e_1]\!]^{\mathsf{A}}_{\mathcal{T}}\{(v\wedge\sigma,\sigma')\}) \cup ([\![e_2]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\neg v\wedge\sigma,\sigma')\}) \end{array}$$

$$\frac{[\![e]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} = \{(\sigma_1,\sigma_1')\} \quad \rho = [x_1/x', r_1/\mathtt{res}] \quad \mathit{fresh\ logical}\ x_1, r_1}{[\![x := e]\!]^{\mathsf{A}}_{\mathcal{T}}\{(\sigma,\sigma')\} =_{df} \{((\rho\sigma_1) \wedge x'{=}r_1, \rho\sigma_1')\}}$$

$$\frac{\sigma \vdash \Phi_{pr} * \sigma_{\mathtt{R}} \text{ and } \sigma_1'{=}\mathtt{emp},\ \text{or } \sigma * [\sigma_1'] \rhd \Phi_{pr} * \sigma_{\mathtt{R}} \text{ for each } (\sigma,\sigma')}{[\![\mathtt{while}\ x\ \{e\}\ \mathtt{where}\ \Phi_{pr} \ast\!\!\rightarrow \Phi_{po}]\!]_{\mathcal{T}}\{(\sigma,\sigma')\} =_{df} \{(\sigma_{\mathtt{R}} * \sigma_{po} \wedge \neg x, \sigma' * \sigma_1') \mid \sigma_{po} \in \Phi_{po}\}}$$

which is used in both the first and the third verification algorithms.

6.5.5 Soundness

Informally, in the presence of unknown procedure calls, the soundness of the verification signifies that, a program is successfully verified against its specifications, if all the unknown procedures that it invokes conform to the specifications discovered by the verification algorithm. Therefore, the correctness of the program depends on a (possible) further verification for the unknown procedures. It can be defined as follows:

Definition 6.5.1 (Soundness) *Suppose that for specification table $\mathcal{T}$, program to be verified $v = \{e_1; u; e_2\}$ and its specifications $mspec_v$, our verification succeeds and returns $\mathcal{T}_u$ as the specification table for unknown procedures invoked in v. We say our verification is sound, if the following holds:*

$$\forall \sigma \in [\![e_1; u; e_2]\!]_{\mathcal{T} \uplus \mathcal{T}_U}([\boldsymbol{x_0}/\boldsymbol{a}, \boldsymbol{y_0}/\boldsymbol{b}]\Phi_{pr}) \cdot \sigma \vdash [\boldsymbol{x_0}/\boldsymbol{a}, \boldsymbol{y_0}/\boldsymbol{b}, \boldsymbol{y'_0}/\boldsymbol{b'}]\Phi_{po} * \texttt{true}$$

which means that, with respect to the underlying semantics, if all the unknown procedures can be verified to satisfy their specifications in $\mathcal{T}_u$, then the whole program v should meet all the specifications in $mspec_v$.

The soundness of our verification algorithm is guaranteed with several aspects: the soundness of the entailment checking, the soundness of our abduction, and the soundness of our abstract semantics. The proof for entailment checking is by structural induction over abstract domain (Nguyen et al., 2007). For abduction, as its rules show, the abduction result σ' is always checked together with the antecedent σ such that they can entail the consequence, and hence its soundness follows that of entailment checking's. Finally, the soundness of abstract semantics is proven by induction over program constructors. Therefore we have

Theorem 6.5.2 (Soundness) *Our verification of programs with partially available code is sound.*

Detailed proof can be found in Appendix A.

6.6 Summary

It is a both practical and challenging problem to verify both memory safety and functional correctness of heap-manipulating imperative programs with unknown procedure calls. We propose an approach to solving it by inferring expected specifications for unknown procedures from their calling contexts. Then the program is proven correct on condition that the invoked unknown procedures meet the inferred specifications. We employ a forward program analysis over a combined domain and invent a novel abduction for it to synthesise the specifications of the unknown procedure. As a proof of concept, we have also implemented a system to test the viability of the proposed approach, whose results will be discussed in the next chapter. Our main future work is to explore more general solution for unknown calls in sequence to achieve more reasonable specifications for them.

Chapter 7

Experiments and Evaluation

This chapter presents the experimental results from the systems we implemented as a proof of theory, and the evaluations of this thesis' work according to the criteria proposed in Chapter 1.

7.1 Experimental Results

7.1.1 Partial Specification Refinement

We have implemented a verification system with Objective Caml to evaluate our approach to partial specification refinement. Our experimental results were achieved with an Intel Core 2 CPU 2.66GHz with 8Gb RAM. The four columns in Table 7.1 and Table 7.2 describe respectively the analysed programs, the analysis time in seconds, and the methods' (given and inferred) preconditions and postconditions. All formulae with a grey background are inferred by our analysis. For some programs,

Prog.	**Time**	**Pre**	**Post**
create*	0.379	emp ∧ n≥0	res::llB⟨S⟩ ∧ n=|S| ∧ ∀v∈S·1≤v≤n
	1.752	emp ∧ n≥0	res::dllB⟨r,S⟩∧ n=|S|∧∀v∈S·1≤v≤n
	0.954	emp ∧ n≥0	res::sllB2⟨S⟩∧ n=|S|∧∀v∈S·1≤v≤n
sort_* insert	0.591	x::ll⟨n⟩ ∧ n≥1	x::ll⟨m⟩ ∧ m=n+1
	0.789	x::dll⟨p, n⟩ ∧ n≥1	x::dll⟨q, m⟩ ∧ n≥1 ∧ m=n+1 ∧ p=q
	0.504	x::sll⟨n, xs, xl⟩∧ v≥xs	x::sll⟨m, mn, mx⟩ ∧ xs=mn ∧ mx=max(xl,v)∧m=n+1
tail_ insert	0.566	x::ll⟨n⟩ ∧ n≥1	x::ll⟨m⟩ ∧ m=n+1
	0.628	x::sll⟨n, xs, xl⟩∧ v≥xl	x::sll⟨m, mn, mx⟩ ∧ v=mx∧ mn=xs ∧ m=n+1
rand_* insert	0.522	x::ll⟨n⟩ ∧ n≥1	x::ll⟨m⟩ ∧ m=n+1
	0.830	x::dll⟨p, n⟩ ∧ n≥1	x::dll⟨q, m⟩ ∧ m=n+1 ∧ p=q
	—	x::sll⟨n, xs, xl⟩∧ (fail)	x::sll⟨m, mn, mx⟩∧ (fail)
delete	0.630	x::llB⟨S⟩ ∧ |S|≥2	x::llB⟨T⟩ ∧ ∃a.S=T⊔{a}
	1.024	x::sllB⟨S⟩ ∧ |S|≥2	x::sllB⟨T⟩ ∧ ∃a.S=T⊔{a}
delete	1.252	x::dllB⟨p, S⟩ ∧ |S|≥2	x::dllB⟨q, T⟩ ∧ ∃a.S=T⊔{a} ∧ p=q
travrs	0.296	x::ll⟨m⟩∧ n≥0∧m≥n	x::ls⟨p, k⟩ ∗ res::ll⟨r⟩ ∧ p=res∧ k=n ∧ m=n+r
	2.205	x::sllB⟨S⟩ ∧ n≥0∧|S|≥n	x::slsB⟨p, T⟩ ∗ res::sllB⟨S_2⟩∧ p=res ∧|T|=n∧S=T⊔S_2∧∀u∈T,v∈S_2·u≤v
append*	0.512	x::ll⟨xn⟩∗y::ll⟨yn⟩∧ xn≥1	x::ll⟨m⟩ ∧ m=xn+yn
	0.660	x::dll⟨xp, xn⟩ ∗ y::dll⟨yp, yn⟩ ∧ xn≥1	x::dll⟨q, m⟩ ∧ m=xn+yn ∧ q=xp
	0.948	x::sll⟨xn, xs, xl⟩∧ xl≤ys ∗ y::sll⟨yn, ys, yl⟩	x::sll⟨m, rs, rl⟩ ∧ yl=rl ∧ m≥1+yn ∧ m=xn+yn
dispatch3	0.786	lst::ll2⟨n, s⟩	gtl′::ll2⟨n_1,s_1⟩∗ltl′::ll2⟨n_2,s_2⟩∧ n= n_1+n_2∧s=s_1+s_2∧s_1≥3n_1∧s_2<3n_2+1

Table 7.1: Experimental results for list-processing programs.

Prog.	Time	Pre	Post
travrs	0.532	x::bt⟨S, h⟩	x::bt⟨T, k⟩ ∧ S=T ∧ h=k
count	0.709	x::bt⟨S, h⟩	x::bt⟨T, k⟩ ∧ res=|S| ∧ S=T ∧ h=k
height	0.913	x::bt⟨S, h⟩	x::bt⟨T, k⟩ ∧ res=h=k ∧ S=T
insert	1.276	x::bt⟨S, h⟩ ∧ |S|≥1 ∧ h≥1	x::bt⟨T, k⟩ ∧ T=S⊔{v} ∧ h≤k≤h+1
delete	0.970	x::bt⟨S, h⟩ ∧ |S|≥2 ∧ h≥2	x::bt⟨T, k⟩∧ ∃a·S=T⊔{a}∧h−1≤k≤h
search	1.583	x::bst⟨sm, lg⟩	x::bst⟨mn, mx⟩ ∧ sm=mn ∧ lg=mx ∧ 0≤res≤1
bst_ insert	1.720	x::bst⟨sm, lg⟩	x::bst⟨mn, mx⟩ ∧ (v<sm∧v=mn ∧ lg=mx ∨ lg<v∧v=mx∧sm=mn ∨ sm=mn∧lg=mx)
avl_ins	11.12	x::avl⟨S, h⟩	res::avl⟨T, k⟩∧ T=S⊔{v}∧h≤k≤h+1
sdl2nbt	5.826	x::sdlB⟨p,q,S⟩∧ |S|≥1∧ p=null ∧ q=tail	res::nbt⟨T⟩ ∧ T=S

Table 7.2: Experimental results for tree-processing programs.

we have verified them with different pre/post shape templates. Programs with star ∗ have different versions for various data structures.

The results highlight the refinement of both pre- and postconditions based on user-provided shape specifications, even for complicated data structure such as AVL trees. Firstly, our approach can compute non-trivial pure constraints for postcondition. For example, the program `create`'s code is listed as follows:

```
Node create(int n) {
  if (n == 0) return null;
  else {
    Node r = create(n - 1);
    return new Node(n, r);
  }
```

```
}
```

After the refinement we obtain the value range in the created list (between one and `n`). For `delete` which removes a random node from the list, we know the content of the result list is subsumed by that of the input list. For a sample program `dispatch3` from Bouajjani et al. (2010) which divides a list into two lists where one only contain values no less than three and the other less than three, we obtain a detailed quantitative relationship $n{=}n_1{+}n_2{\wedge}s{=}s_1{+}s_2{\wedge}s_1{\geq}3n_1{\wedge}s_2{<}3n_2{+}1$. For list-sorting algorithms, we confirm the content of the output is the same as that of the input. For tree-processing programs (`insert`, `delete` and `avl_ins`), we obtain that the height difference between the input and output trees is at most one. Meanwhile, we can calculate non-trivial requirements in precondition for memory safety or functional correctness. For many programs we can get the constraints for precondition such that the input list/tree should have at least one (or two) nodes for the sake of memory safety. A more interesting example is the `travrs` method which takes in a list with length `m` and an integer `n` and then traverses towards the tail of the list for `n` steps:

```
Node travrs(Node x, int n) {
  if (n == 0) return x;
  else return travrs(x.next, n - 1);
}
```

For this program, the analysis discovers $m{\geq}n$ in the precondition to ensure memory safety. Another example is the `append` method concatenating two sorted lists into one:

```
void append(Node x, Node y) {
```

```
  Node w = x.next;
  if (w == null) x.next = y;
  else append(w, y);
}
```

To ensure that the result list is sorted, the analysis figures out that the minimum value in the second list must be no less than the maximum value in the first list. Without those discovered constraints the program will either cause memory violation or cannot meet the specification.

A second highlight is our flexibility by supporting multiple predicates. Our analysis tries to refine different specifications for the same program at various correctness levels (with different predicates), for instance `sort_insert`, `tail_insert` and `append`. For `rand_insert`, which inserts a node into a random place (after the head) of a list, we confirm that the list's length is increased by one, but cannot verify the list is kept sorted if it was before the insertion, as the result indicates.

We have also tried our approach over part of the FreeRTOS kernel (Barry, 2006). For its list processing programs `list.h` and `list.c` (472 lines with intensive manipulation over composite sorted doubly-linked lists) it took 2.85 seconds for our system to refine all the specifications given for the main functions, which further confirms the viability of our approach.

Prog.	Time	Pre	Post
create (main)		emp ∧ n≥0	res::sllB2⟨S⟩∧ n=|S|∧∀v∈S·1≤v≤n
alloc	0.214	emp	res::Node⟨n, r⟩
travrs (main)		x::slsB⟨p, T⟩ ∗ res::sllB⟨S_2⟩∧ p=res ∧|T|=n∧S=T⊔S_2∧∀u∈T,v∈S_2·u≤v	
next	0.135	x::sllB⟨S⟩ ∧ |S|≥0	x::Node⟨v, res⟩ ∗ res::sllB⟨S_1⟩ ∧ S={v}⊔S_1
append (main)		x::ll⟨xn⟩∗y::ll⟨yn⟩∧ xn≥1	x::ll⟨m⟩ ∧ m=xn+yn
travrs	0.619	x::ll⟨n⟩ ∧ n≥1	x::ls⟨res, m⟩ ∗ res::Node⟨v, null⟩
Sorting (main)		x::llB⟨S⟩ ∧ |S|≥1	res::sllB⟨T⟩ ∧ T=S (‡)
merge	4.107	x::sllB⟨S_x⟩ ∗ y::sllB⟨S_y⟩	res::sllB⟨T⟩ ∧ T=S_x⊔S_y
flatten	2.693	x::bstB⟨S⟩	res::sllB⟨T⟩ ∧ T=S
insert	0.824	r::sllB⟨S⟩ ∗ x::Node⟨v, _⟩	res::sllB⟨T⟩ ∧ T=S⊔{v}

Table 7.3: Experimental results for list-processing programs.

7.1.2 Specification Synthesis for Auxiliary Methods and Loops

For specification synthesis, we have extended the system in the previous section for evaluation purpose. Our experimental results were achieved with the same hardware environment. The four columns in Table 7.3 describe respectively the analysed programs, the analysis time in seconds, and the methods' (given and inferred) preconditions and postconditions. The programs whose names have (main) denote the primary procedures (without a timing report as most of them are already shown in the previous section). The programs with time information exhibit the auxiliary procedures whose specifications are wholly synthesised and inferred (as their background is totally grey).

For the experimental results, it can be seen that our approach effectively reduces

Prog.	Time	Pre	Post
bst_insert	(main)	x::bst⟨sm, lg⟩	x::bst⟨mn, mx⟩ ∧ (v<sm∧v=mn ∧ lg=mx ∨ lg<v∧v=mx∧sm=mn ∨ sm=mn∧lg=mx)
alloc	0.255	emp	res::Node2⟨v, null, null⟩
avl_ins	(main)	x::avl⟨S, h⟩	res::avl⟨T, k⟩∧ T=S⊔{v}∧h≤k≤h+1
height	0.775	x::avl⟨S, h⟩	x::avl⟨T, k⟩ ∧ T=S ∧ h=k=res
sdl2nbt	(main)	x::sdlB⟨p,q,S⟩∧ |S|≥1∧ p=null ∧ q=tail	res::nbt⟨T⟩ ∧ T=S
loop1	0.496	head::sdlB⟨p,q,S⟩∧ p=null ∧ q=tail∧ head=root=end	head::sdlB⟨null, q, S_h⟩ ∧ end=tail∗ root::sdlB⟨p, tail, S_r⟩ ∧ S=S_h⊔S_r∧ (∀x∈S_h, y∈S_r·x≤y) ∧ 0≤|S_h|−|S_r|≤1

Table 7.4: Experimental results for tree-processing programs.

user annotations by synthesising specifications for auxiliary methods, given raw specifications of primary methods. For example, we have manually set some of the instructions in the programs in the previous section as auxiliary procedures and tried to generate their specifications, such as `alloc`, `next` and `travrs` in list-processing programs and `alloc` for `bst_insert`. For new test suites, we have analysed a number of list-sorting algorithms with at least one auxiliary method each. Note that all the list-sorting algorithms have the same specification for their primary methods (line ‡), while the annotations found by our approach for auxiliary methods are diverse (the grey lines below ‡). From the automatically derived specifications, we can see that the auxiliary method `merge` for `merge_sort` merges two sorted lists into one, and the auxiliary method `flatten` for `tree_sort` flattens a binary search tree into a sorted list. As another example, `avl_ins` also has some auxiliary (recursive) methods such as calculation of tree's height and double rotation, which are automatically analysed as well.

One observation over the experimental results is that some of the derived specifica-

tions are quite complex if written by hand, for example `loop1`. If we require users to provide these specifications for auxiliary methods (even shape-only), it is still quite tedious and error-prone. On the contrary, the users now have the option to leave such work for our system, which witnesses the value of our improvement of the verification process.

7.1.3 Verification of Programs with Unknown Components

For the verification of program with unknown components, we have implemented the verification algorithms and the abstract semantics with Objective Caml and evaluated them over some heap-manipulating programs. The results are in Table 7.5, Table 7.6, Table 7.7, Table 7.8 and Table 7.9. In each table, the first and second columns denote the programs used for evaluation and their time consumption, respectively. During the experiments, we manually hide some instructions in the original programs as calls to unknown procedures, whose specifications we try to discover during the verification process. Accordingly, the third column in the first two tables contain both the specifications of the programs to be verified (upper line), and the derived specifications for the unknown procedure (lower line). For the third table, as we use the same specification $\texttt{x}{::}\texttt{llB}\langle\texttt{S}\rangle \mathrel{*\!\!\!\rightarrow} \texttt{res}{::}\texttt{sllB}\langle\texttt{T}\rangle \wedge \texttt{T}{=}\texttt{S}$ to verify all the sorting algorithms, the third column (from the second line on) states the discovered specification for the unknown call only. Similar as previous examples, some of the programs with the same name have different versions, say, the ones processing (sorted) singly-linked lists (`ll` and `sll`) are different from their counterparts for doubly-linked lists (`dll`).

It can be seen that all programs are successfully verified, with some obligations on the unknown calls discovered. We note down two observations on the experimental results. The first is that the discovered specifications for the unknown procedures

Prog.	**Time**	**Main spec.** ($\Phi_{pr} \ast\!\!\rightarrow \Phi_{po}$) & **Derived unknown spec.** ($\Phi^u_{pr} \ast\!\!\rightarrow \Phi^u_{po}$)
create*	0.405	$\texttt{emp} \wedge \texttt{n}{\geq}0 \ast\!\!\rightarrow \texttt{res::llB}\langle \texttt{S}\rangle \wedge \texttt{n}{=}|\texttt{S}| \wedge \forall \texttt{v}{\in}\texttt{S}{\cdot}1{\leq}\texttt{v}{\leq}\texttt{n}$
		$\texttt{emp} \wedge \texttt{a}{\geq}1 \ \ast\!\!\rightarrow \texttt{res::Node}\langle \texttt{c}, \texttt{b}\rangle \wedge 1{\leq}\texttt{c}{\leq}\texttt{n}$
	1.895	$\texttt{emp} \wedge \texttt{n}{\geq}0 \ \ast\!\!\rightarrow \texttt{res::dllB}\langle \texttt{rp}, \texttt{S}\rangle \wedge \texttt{n}{=}|\texttt{S}| \wedge \forall \texttt{v}{\in}\texttt{S}{\cdot}1{\leq}\texttt{v}{\leq}\texttt{n}$
		$\texttt{emp} \wedge \texttt{a}{\geq}1 \ \ast\!\!\rightarrow \texttt{res::Node2}\langle \texttt{c}, \texttt{d}, \texttt{b}\rangle \wedge 1{\leq}\texttt{c}{\leq}\texttt{n}$
	1.020	$\texttt{emp} \wedge \texttt{n}{\geq}0 \ \ast\!\!\rightarrow \texttt{res::sllB2}\langle \texttt{S}\rangle \wedge \texttt{n}{=}|\texttt{S}| \wedge \forall \texttt{v}{\in}\texttt{S}{\cdot}1{\leq}\texttt{v}{\leq}\texttt{n}$
		$\texttt{emp} \wedge \texttt{a}{\geq}1 \ \ast\!\!\rightarrow \texttt{res::Node}\langle \texttt{c}, \texttt{b}\rangle \wedge \texttt{a}{-}1{\leq}\texttt{c}{\leq}\texttt{a}$
sort_* insert	0.667	$\texttt{x::ll}\langle \texttt{n}\rangle \wedge \texttt{n}{\geq}1 \ \ast\!\!\rightarrow \texttt{x::ll}\langle \texttt{m}\rangle \wedge \texttt{m}{=}\texttt{n}{+}1$
		$\texttt{a::Node}\langle \texttt{b}, \texttt{c}\rangle * \texttt{c::ll}\langle \texttt{d}\rangle \ \ast\!\!\rightarrow \texttt{a::Node}\langle \texttt{b}, \texttt{e}\rangle * \texttt{e::ll}\langle \texttt{d}{+}1\rangle$
	0.842	$\texttt{x::dll}\langle \texttt{p}, \texttt{n}\rangle \wedge \texttt{n}{\geq}1 \ \ast\!\!\rightarrow \texttt{x::dll}\langle \texttt{q}, \texttt{m}\rangle \wedge \texttt{n}{\geq}1 \wedge \texttt{m}{=}\texttt{n}{+}1 \wedge \texttt{p}{=}\texttt{q}$
		$\texttt{a::Node2}\langle \texttt{b}, \texttt{g}, \texttt{c}\rangle * \texttt{c::dll}\langle \texttt{a}, \texttt{d}\rangle \ \ast\!\!\rightarrow \texttt{a::Node2}\langle \texttt{b}, \texttt{g}, \texttt{e}\rangle * \texttt{e::dll}\langle \texttt{a}, \texttt{d}{+}1\rangle$
	0.764	$\texttt{x::sll}\langle \texttt{n,xs,xl}\rangle \wedge \texttt{v}{\geq}\texttt{xs} \ \ast\!\!\rightarrow$ $\texttt{x::sll}\langle \texttt{n}{+}1\texttt{,mn,mx}\rangle \wedge \texttt{mn}{=}\texttt{xs} \wedge \texttt{mx}{=}\texttt{max}(\texttt{xl,v})$
		$\texttt{a::Node}\langle \texttt{b,c}\rangle * \texttt{c::sll}\langle \texttt{d,g,h}\rangle \wedge \texttt{b}{\leq}\texttt{f}{\leq}\texttt{g} \ \ast\!\!\rightarrow \texttt{a::Node}\langle \texttt{b,e}\rangle * \texttt{e::sll}\langle \texttt{d}{+}1\texttt{,f,h}\rangle$
tail_ insert	0.498	$\texttt{x::ll}\langle \texttt{n}\rangle \wedge \texttt{n}{\geq}1 \ \ast\!\!\rightarrow \texttt{x::ll}\langle \texttt{m}\rangle \wedge \texttt{m}{=}\texttt{n}{+}1$
		$\texttt{a::Node}\langle \texttt{b}, \texttt{null}\rangle \ \ast\!\!\rightarrow \texttt{a::ll}\langle 2\rangle$
	0.627	$\texttt{x::sll}\langle \texttt{n}, \texttt{xs}, \texttt{xl}\rangle \wedge \texttt{v}{\geq}\texttt{xl} \ \ast\!\!\rightarrow$ $\texttt{x::sll}\langle \texttt{m}, \texttt{mn}, \texttt{mx}\rangle \wedge \texttt{v}{=}\texttt{mx} \wedge \texttt{mn}{=}\texttt{xs} \wedge \texttt{m}{=}\texttt{n}{+}1$
		$\texttt{a::Node}\langle \texttt{b}, \texttt{null}\rangle \wedge \texttt{b}{\leq}\texttt{c} \ \ast\!\!\rightarrow \texttt{a::sll}\langle 2, \texttt{b}, \texttt{c}\rangle$
rand_* insert	0.514	$\texttt{x::ll}\langle \texttt{n}\rangle \wedge \texttt{n}{\geq}1 \ \ast\!\!\rightarrow \texttt{x::ll}\langle \texttt{m}\rangle \wedge \texttt{m}{=}\texttt{n}{+}1$
		$\texttt{a::Node}\langle \texttt{b}, \texttt{c}\rangle * \texttt{c::ll}\langle \texttt{d}\rangle \ \ast\!\!\rightarrow \texttt{a::Node}\langle \texttt{b}, \texttt{e}\rangle * \texttt{e::ll}\langle \texttt{d}{+}1\rangle$
	0.697	$\texttt{x::dll}\langle \texttt{p}, \texttt{n}\rangle \wedge \texttt{n}{\geq}1 \ \ast\!\!\rightarrow \texttt{x::dll}\langle \texttt{q}, \texttt{m}\rangle \wedge \texttt{m}{=}\texttt{n}{+}1 \wedge \texttt{p}{=}\texttt{q}$
		$\texttt{a::Node2}\langle \texttt{b}, \texttt{g}, \texttt{c}\rangle * \texttt{c::dll}\langle \texttt{a}, \texttt{d}\rangle \ \ast\!\!\rightarrow \texttt{a::Node2}\langle \texttt{b}, \texttt{g}, \texttt{e}\rangle * \texttt{e::dll}\langle \texttt{a}, \texttt{d}{+}1\rangle$

Table 7.5: Experimental results (lists, part 1).

Prog.	**Time**	**Main spec.** ($\Phi_{pr} \ast\!\!\rightarrow \Phi_{po}$) and **Derived unknown spec.** ($\Phi^u_{pr} \ast\!\!\rightarrow \Phi^u_{po}$)
delete*	0.646	$\texttt{x::llB}\langle\texttt{S}\rangle \wedge \texttt{|S|}{\geq}\texttt{2} \ \ast\!\!\rightarrow\ \texttt{x::llB}\langle\texttt{T}\rangle \wedge \exists\texttt{a}\cdot\texttt{S}{=}\texttt{T}\sqcup\{\texttt{a}\}$
		$\texttt{a::Node}\langle\texttt{b},\texttt{c}\rangle \ast \texttt{c::Node}\langle\texttt{d},\texttt{e}\rangle \ast \texttt{e::llB}\langle\texttt{E}\rangle \ \ast\!\!\rightarrow \texttt{a::Node}\langle\texttt{b},\texttt{e}\rangle \ast \texttt{e::llB}\langle\texttt{E}\rangle$
	0.916	$\texttt{x::sllB}\langle\texttt{S}\rangle \wedge \texttt{|S|}{\geq}\texttt{2} \ \ast\!\!\rightarrow\ \texttt{x::sllB}\langle\texttt{T}\rangle \wedge \exists\texttt{a}\cdot\texttt{S}{=}\texttt{T}\sqcup\{\texttt{a}\}$
		$\texttt{a::Node}\langle\texttt{b},\texttt{c}\rangle \ast \texttt{c::Node}\langle\texttt{d},\texttt{e}\rangle \ast \texttt{e::sllB}\langle\texttt{E}\rangle \wedge \forall\texttt{f}{\in}\texttt{E}\cdot\texttt{b}{\leq}\texttt{d}{\leq}\texttt{f} \ \ast\!\!\rightarrow$ $\texttt{a::Node}\langle\texttt{b},\texttt{e}\rangle \ast \texttt{e::sllB}\langle\texttt{E}\rangle \wedge \forall\texttt{f}{\in}\texttt{E}\cdot\texttt{b}{\leq}\texttt{f}$
	1.430	$\texttt{x::dllB}\langle\texttt{p},\texttt{S}\rangle \wedge \texttt{|S|}{\geq}\texttt{2} \ \ast\!\!\rightarrow\ \texttt{x::dllB}\langle\texttt{q},\texttt{T}\rangle \wedge \exists\texttt{a}\cdot\texttt{S}{=}\texttt{T}\sqcup\{\texttt{a}\} \wedge \texttt{p}{=}\texttt{q}$
		$\texttt{a::Node2}\langle\texttt{b},\texttt{f},\texttt{c}\rangle \ast \texttt{c::Node}\langle\texttt{d},\texttt{a},\texttt{e}\rangle \ast \texttt{e::dllB}\langle\texttt{c},\texttt{E}\rangle \ \ast\!\!\rightarrow$ $\texttt{a::Node}\langle\texttt{b},\texttt{f},\texttt{e}\rangle \ast \texttt{e::dllB}\langle\texttt{a},\texttt{E}\rangle$
append	0.523	$\texttt{x::ll}\langle\texttt{xn}\rangle \ast \texttt{y::ll}\langle\texttt{yn}\rangle \wedge \texttt{xn}{\geq}\texttt{1} \ \ast\!\!\rightarrow\ \texttt{x::ll}\langle\texttt{m}\rangle \wedge \texttt{m}{=}\texttt{xn}{+}\texttt{yn}$
		$\texttt{a::ll}\langle\texttt{b}\rangle \ \ast\!\!\rightarrow \texttt{a::ls}\langle\texttt{res},\texttt{c}\rangle \ast \texttt{res::ll}\langle\texttt{d}\rangle \wedge \texttt{b}{=}\texttt{c}{+}\texttt{d}$
	0.861	$\texttt{x::sll}\langle\texttt{xn},\texttt{xs},\texttt{xl}\rangle \ast \texttt{y::sll}\langle\texttt{yn},\texttt{ys},\texttt{yl}\rangle \wedge \texttt{xl}{\leq}\texttt{ys} \ \ast\!\!\rightarrow$ $\texttt{x::sll}\langle\texttt{m},\texttt{rs},\texttt{rl}\rangle \wedge \texttt{yl}{=}\texttt{rl} \wedge \texttt{m}{\geq}\texttt{1}{+}\texttt{yn} \wedge \texttt{m}{=}\texttt{xn}{+}\texttt{yn}$
		$\texttt{a::sllB}\langle\texttt{A}\rangle \ \ast\!\!\rightarrow \texttt{a::slsB}\langle\texttt{A}_1\rangle \ast \texttt{res::sllB}\langle\texttt{R}\rangle \wedge \texttt{A}{=}\texttt{A}_1{\sqcup}\texttt{R} \wedge \forall\texttt{b}{\in}\texttt{A}_1,\texttt{c}{\in}\texttt{R}\cdot\texttt{b}{\leq}\texttt{c}$

Table 7.6: Experimental results (lists, part 2).

are usually more general than what we expect. Bear in mind that we have replaced some instructions from those programs with unknown calls. We have compared the inferred specifications for those unknown calls with the original instructions. The results show that the specifications derived by our algorithm not only fully capture the behaviours of those instructions, but also suggest other possible implementations. A case in point is list's `append`:

```
void append(Node x, Node y) {
  Node w = unknown(x);
  if (w.next == null) w.next = y;
  else append(w.next, y);
}
```

As can be seen in the code, its "unknown call" was originally an evaluation of `x`.

Prog.	**Time**	**Main spec.** ($\Phi_{pr} \ast\!\!\rightarrow \Phi_{po}$) and **Derived unknown spec.** ($\Phi^u_{pr} \ast\!\!\rightarrow \Phi^u_{po}$)
Binary tree processing programs		
`travrs`	0.417	$\texttt{x::bt}\langle \texttt{S},\texttt{h}\rangle \ \ast\!\!\rightarrow \texttt{x::bt}\langle \texttt{T},\texttt{k}\rangle \wedge \texttt{S=T} \wedge \texttt{h=k}$
		$\texttt{a::bt}\langle \texttt{A},\texttt{b}\rangle \wedge \texttt{a}\neq\texttt{null} \ \ast\!\!\rightarrow \texttt{a::Node2}\langle \texttt{c},\texttt{d}',\texttt{e}'\rangle \ast \texttt{d}'\texttt{::bt}\langle \texttt{D},\texttt{f}\rangle \ast \texttt{e}'\texttt{::bt}\langle \texttt{E},\texttt{g}\rangle \wedge \texttt{A=}\{\texttt{c}\}\sqcup\texttt{D}\sqcup\texttt{E} \wedge \texttt{b=max(f,g)+1}$, or $\texttt{a::bt}\langle \texttt{A},\texttt{b}\rangle \wedge \texttt{a}\neq\texttt{null} \ \ast\!\!\rightarrow \texttt{a::Node2}\langle \texttt{c},\texttt{e}',\texttt{d}'\rangle \ast \texttt{d}'\texttt{::bt}\langle \texttt{D},\texttt{f}\rangle \ast \texttt{e}'\texttt{::bt}\langle \texttt{E},\texttt{g}\rangle \wedge \texttt{A=}\{\texttt{c}\}\sqcup\texttt{D}\sqcup\texttt{E} \wedge \texttt{b=max(f,g)+1}$
`count`	0.705	$\texttt{x::bt}\langle \texttt{S},\texttt{h}\rangle \ \ast\!\!\rightarrow \texttt{x::bt}\langle \texttt{T},\texttt{k}\rangle \wedge \texttt{res=}|\texttt{S}| \wedge \texttt{S=T} \wedge \texttt{h=k}$
		$\texttt{a::bt}\langle \texttt{A},\texttt{b}\rangle \wedge \texttt{a}\neq\texttt{null} \ \ast\!\!\rightarrow \texttt{a::Node2}\langle \texttt{c},\texttt{d}',\texttt{e}'\rangle \ast \texttt{d}'\texttt{::bt}\langle \texttt{D},\texttt{f}\rangle \ast \texttt{e}'\texttt{::bt}\langle \texttt{E},\texttt{g}\rangle \wedge \texttt{A=}\{\texttt{c}\}\sqcup\texttt{D}\sqcup\texttt{E} \wedge \texttt{b=max(f,g)+1}$, or $\texttt{a::bt}\langle \texttt{A},\texttt{b}\rangle \wedge \texttt{a}\neq\texttt{null} \ \ast\!\!\rightarrow \texttt{a::Node2}\langle \texttt{c},\texttt{e}',\texttt{d}'\rangle \ast \texttt{d}'\texttt{::bt}\langle \texttt{D},\texttt{f}\rangle \ast \texttt{e}'\texttt{::bt}\langle \texttt{E},\texttt{g}\rangle \wedge \texttt{A=}\{\texttt{c}\}\sqcup\texttt{D}\sqcup\texttt{E} \wedge \texttt{b=max(f,g)+1}$
`height`	0.821	$\texttt{x::bt}\langle \texttt{S},\texttt{h}\rangle \ \ast\!\!\rightarrow \texttt{x::bt}\langle \texttt{T},\texttt{k}\rangle \wedge \texttt{res=h=k} \wedge \texttt{S=T}$
		$\texttt{a::bt}\langle \texttt{A},\texttt{b}\rangle \wedge \texttt{a}\neq\texttt{null} \ \ast\!\!\rightarrow \texttt{a::Node2}\langle \texttt{c},\texttt{d}',\texttt{e}'\rangle \ast \texttt{d}'\texttt{::bt}\langle \texttt{D},\texttt{f}\rangle \ast \texttt{e}'\texttt{::bt}\langle \texttt{E},\texttt{g}\rangle \wedge \texttt{A=}\{\texttt{c}\}\sqcup\texttt{D}\sqcup\texttt{E} \wedge \texttt{b=max(f,g)+1}$, or $\texttt{a::bt}\langle \texttt{A},\texttt{b}\rangle \wedge \texttt{a}\neq\texttt{null} \ \ast\!\!\rightarrow \texttt{a::Node2}\langle \texttt{c},\texttt{e}',\texttt{d}'\rangle \ast \texttt{d}'\texttt{::bt}\langle \texttt{D},\texttt{f}\rangle \ast \texttt{e}'\texttt{::bt}\langle \texttt{E},\texttt{g}\rangle \wedge \texttt{A=}\{\texttt{c}\}\sqcup\texttt{D}\sqcup\texttt{E} \wedge \texttt{b=max(f,g)+1}$
`insert`	1.354	$\texttt{x::bt}\langle \texttt{S},\texttt{h}\rangle \wedge |\texttt{S}|\geq 1 \wedge \texttt{h}\geq 1 \ \ast\!\!\rightarrow \texttt{x::bt}\langle \texttt{T},\texttt{k}\rangle \wedge \texttt{T=S}\sqcup\{\texttt{v}\} \wedge \texttt{h}\leq\texttt{k}\leq\texttt{h+1}$
		$\texttt{a::Node2}\langle \texttt{b},\texttt{null},\texttt{null}\rangle \ \ast\!\!\rightarrow \texttt{a::bt}\langle \texttt{A},2\rangle \wedge \texttt{A=}\{\texttt{b},\texttt{c}\}$
`delete`	1.019	$\texttt{x::bt}\langle \texttt{S},\texttt{h}\rangle \wedge |\texttt{S}|\geq 2 \wedge \texttt{h}\geq 2 \ \ast\!\!\rightarrow \texttt{x::bt}\langle \texttt{T},\texttt{k}\rangle \wedge \exists \texttt{a} \cdot \texttt{S=T}\sqcup\{\texttt{a}\} \wedge \texttt{h}-1\leq\texttt{k}\leq\texttt{h}$
		$\texttt{a::Node2}\langle \texttt{b,c,null}\rangle \ast \texttt{c::Node2}\langle \texttt{d,null,null}\rangle \ast\!\!\rightarrow \texttt{a::Node2}\langle \texttt{b,null,null}\rangle$, & $\texttt{a::Node2}\langle \texttt{b,null,c}\rangle \ast \texttt{c::Node2}\langle \texttt{d,null,null}\rangle \ast\!\!\rightarrow \texttt{a::Node2}\langle \texttt{b,null,null}\rangle$

Table 7.7: Experimental results (trees, part 1).

Prog.	**Time**	**Main spec.** ($\Phi_{pr} \ast\!\!\rightarrow \Phi_{po}$) and **Derived unknown spec.** ($\Phi^u_{pr} \ast\!\!\rightarrow \Phi^u_{po}$)
Binary search tree and AVL tree processing programs		
search	1.851	$\texttt{x::bst}\langle \texttt{sm}, \texttt{lg}\rangle \ \ast\!\!\rightarrow \texttt{x::bst}\langle \texttt{mn}, \texttt{mx}\rangle \wedge \texttt{sm}{=}\texttt{mn} \wedge \texttt{lg}{=}\texttt{mx} \wedge 0{\leq}\texttt{res}{\leq}1$
		$\texttt{a::bst}\langle \texttt{b}, \texttt{c}\rangle \wedge \texttt{a}{\neq}\texttt{null} \ast\!\!\rightarrow \texttt{a::Node2}\langle \texttt{d}, \texttt{e}', \texttt{f}'\rangle \ast \texttt{e}'\texttt{::bst}\langle \texttt{b}, \texttt{g}\rangle \ast \texttt{f}'\texttt{::bst}\langle \texttt{h}, \texttt{c}\rangle \wedge \texttt{g}{\leq}\texttt{d}{\leq}\texttt{h}$
bst_insert	1.822	$\texttt{x::bst}\langle \texttt{sm}, \texttt{lg}\rangle \ \ast\!\!\rightarrow \texttt{x::bst}\langle \texttt{mn}, \texttt{mx}\rangle \wedge (\texttt{v}{<}\texttt{sm} \wedge \texttt{v}{=}\texttt{mn} \wedge \texttt{lg}{=}\texttt{mx} \vee \texttt{lg}{<}\texttt{v} \wedge \texttt{v}{=}\texttt{mx} \wedge \texttt{sm}{=}\texttt{mn} \vee \texttt{sm}{=}\texttt{mn} \wedge \texttt{lg}{=}\texttt{mx})$
		$\texttt{a::Node2}\langle \texttt{b}, \texttt{null}, \texttt{c}\rangle \ast \texttt{c::bst}\langle \texttt{d}, \texttt{e}\rangle \wedge \texttt{f}{<}\texttt{b}{<}\texttt{d} \ \ast\!\!\rightarrow \texttt{a::bst}\langle \texttt{f}, \texttt{e}\rangle$, and $\texttt{a::Node2}\langle \texttt{b}, \texttt{c}, \texttt{null}\rangle \ast \texttt{c::bst}\langle \texttt{d}, \texttt{e}\rangle \wedge \texttt{e}{<}\texttt{b}{<}\texttt{f} \ \ast\!\!\rightarrow \texttt{a::bst}\langle \texttt{d}, \texttt{f}\rangle$
avl_ins	5.202	$\texttt{x::avl}\langle \texttt{S}, \texttt{h}\rangle \ \ast\!\!\rightarrow \texttt{res::avl}\langle \texttt{T}, \texttt{k}\rangle \wedge \texttt{T}{=}\texttt{S} \sqcup \{\texttt{v}\} \wedge \texttt{h}{\leq}\texttt{k}{\leq}\texttt{h}{+}1$
		$\texttt{a::avl}\langle \texttt{A}, \texttt{b}\rangle \ \ast\!\!\rightarrow \texttt{a::avl}\langle \texttt{A}, \texttt{b}\rangle \wedge \texttt{res}{=}\texttt{b}$
sdl2nbt	5.238	$\texttt{x::sdlB}\langle \texttt{p,q,S}\rangle \wedge \lvert\texttt{S}\rvert{\geq}1 \wedge \texttt{p}{=}\texttt{null} \wedge \texttt{q}{=}\texttt{tail} \ \ast\!\!\rightarrow \texttt{res::nbt}\langle \texttt{T}\rangle \wedge \texttt{T}{=}\texttt{S}$
		$\texttt{a::sdlB}\langle \texttt{null}, \texttt{c}, \texttt{A}\rangle \wedge \texttt{a}{=}\texttt{b}{=}\texttt{d} \ \ast\!\!\rightarrow \texttt{a::sdlB}\langle \texttt{null}, \texttt{b}, \texttt{A}_1\rangle \ast \texttt{b::sdlB}\langle \texttt{e}, \texttt{c}, \texttt{B}\rangle \wedge \texttt{d}{=}\texttt{c} \wedge \texttt{A}{=}\texttt{A}_1 \sqcup \texttt{B} \wedge (\forall \texttt{f}{\in}\texttt{A}_1, \texttt{g}{\in}\texttt{B} \cdot \texttt{f}{\leq}\texttt{g}) \wedge 0{\leq}\lvert\texttt{B}\rvert{-}\lvert\texttt{A}_1\rvert{\leq}1$

Table 7.8: Experimental results (trees, part 2).

Armed with list segment predicates and corresponding lemmas, we are able to infer that the unknown call may actually traverse the list for arbitrary number of nodes, provided it does not go beyond the list's tail. To conclude, our verification always tries its best to find a sound (with respect to the program being verified) and general (with respect to the unknown call) specification for unknown procedures invoked.

The second observation is that the precision of specifications discovered for unknown procedures depends on their callers' given specification. As can be seen we have verified several list-processing programs where each one has various specifications. Within these programs we want to point out that the ones with specifications of both normal lists and sorted lists share the same code (but just with two different specifications). Such examples include `create`, `sort_insert`, `delete`, and so on. For `create`, it creates a list containing numbers from 1 to `n` in descending order:

```
Node create(int n) {
```

Prog.	**Time**	**Main spec.** ($\Phi_{pr} \mathrel{*\!\!\rightarrow} \Phi_{po}$) or **Derived unknown spec.** ($\Phi^u_{pr} \mathrel{*\!\!\rightarrow} \Phi^u_{po}$)
Sorting (main)		$\texttt{x::llB}\langle\texttt{S}\rangle \mathrel{*\!\!\rightarrow} \texttt{res::sllB}\langle\texttt{T}\rangle \wedge \texttt{T=S}$
`merge`	4.099	$\texttt{a::sllB}\langle\texttt{A}\rangle * \texttt{b::sllB}\langle\texttt{B}\rangle \mathrel{*\!\!\rightarrow} \texttt{res::sllB}\langle\texttt{R}\rangle \wedge \texttt{R=A}\sqcup\texttt{B}$
`flatten`	2.680	$\texttt{a::bstB}\langle\texttt{A}\rangle \mathrel{*\!\!\rightarrow} \texttt{res::sllB}\langle\texttt{R}\rangle \wedge \texttt{R=A}$
`insert`	1.667	$\texttt{a::sllB}\langle\texttt{A}\rangle * \texttt{b::Node}\langle\texttt{c},\texttt{d}\rangle \mathrel{*\!\!\rightarrow} \texttt{res::sllB}\langle\texttt{R}\rangle \wedge \texttt{R=A}\sqcup\{\texttt{c}\}$
`unknown`	1.824	$\texttt{a::llB}\langle\texttt{A}\rangle\wedge\texttt{a}\neq\texttt{null} \mathrel{*\!\!\rightarrow} \texttt{res::Node}\langle\texttt{c},\texttt{b}\rangle * \texttt{b::llB}\langle\texttt{B}\rangle\wedge\texttt{A}=\{\texttt{c}\}\sqcup\texttt{B}\wedge\forall\texttt{d}\in\texttt{B}\cdot\texttt{c}\leq\texttt{d}$

Table 7.9: Experimental results (sorting).

```
  if (n == 0) return null;
  else {
    Node r = create(n - 1);
    Node s = unknown(n, r);
    return s;
  }
}
```

We can see from this program that once incorporated with `llB` as specification predicates, the unknown call is expected to return a node whose value `c` is within 1 to `n`. Comparatively, when verified for sortedness, `c` is inferred to be between `a`-1 and `a`, as for sortedness to hold. For `delete`'s sorted version, we also have the extra information that the list with one node removed is still a sorted list (with the multiset value constraints), whose result is stronger than the normal list version.

7.2 Evaluation

The main contribution of this thesis consists of mechanisms for program verification based on only partial information (either specifications or program code). The miss-

ing part of information is inferred with some program analysis techniques making use of information available to the verifier. Our verification system captures program properties over a combined domain of both structural properties and quantitative/-content features, for which we have exploited new techniques such as fixed-point calculation and abductive reasoning over the combined domain. We now evaluate such contributions against the criteria set out in Chapter 1 as follows.

- **For the first objective to allow users to provide only partial specifications for verification, in order to reduce the amount of annotations provided by users, we provide a specification refinement framework to enable the verifier to accept partial specifications.** Under our framework, users are asked to provide only shape information about data structures but not the information on numerical/content part. Our verifier takes over the rest of the work to *refine* the specification by inferring the missing part with an analysis of the program, such that the specification becomes sound with respect to the program being verified. To achieve this, we design a fixed-point computation process for the various program properties which we are interested in. With the help of entailment checking (Chin et al., 2010), the proposed process reduces the fixed-point computation over the combined domain down to the fixed-point calculation over a traditional numerical/content domain which we could deal with existing theorem provers. Meanwhile, for the unsoundness brought in by the incomplete specifications, we develop an abductive reasoning to discover the missing constraints and complete such specifications.

- **Meanwhile, for the first objective, we still stride one step further to reduce even more user-provided specifications by designing a mechanism to infer auxiliary methods' specifications, such that we can verify such methods without annotations.** In such situation where the

methods of a program are divided into primary and auxiliary ones, users may choose to provide partial specifications only for the primary methods. Then we try to infer specifications for the remaining auxiliary methods that the primary methods invoke to allow the verification to continue. The technical contributions include specification synthesis (based on the prior point) and its induced flexible user-guided verification.

- **For the second objective to verify partially available programs with unknown components, we develop a framework to discover necessary obligations for those unknown components.** When the program code is only partially available because of the unknown components, we propose a novel verification framework by inferring a specification for its unknown part from the program contexts, such that the problem of verifying the whole program can be safely reduced to the problem of proving that the unknown part (once its code is available) meets the derived specification. The expected specification for the unknown program part is automatically calculated using an abduction-based shape analysis specifically designed for our combined domain. Meanwhile this approach is essentially *top-down*, as it can be used to infer the specification for callee procedures based on the specification for the caller procedure. Hence it may benefit the general software development process as a complement for current *bottom-up* approaches to verification (Nguyen et al., 2007; Calcagno et al., 2009).

- **For the third objective of experiments over heap-manipulating programs, we have accomplished them with the results presented earlier in this chapter.** From the results it can be seen that all three approaches that we have developed can successfully verify those programs which are correct according to given specifications, and report failures for the programs which will never meet the specifications supplied. The programs cover a wide range of classical algorithms (insertion, deletion, sorting, etc.) for common

data structures (lists and trees, and their sorted and/or balanced versions). Their properties that we verified are also quite rich, from shape information of data structure to numerical information such as size and height, and from relational information like minimum/maximum values, to multiset information as content.

On the basis of the contributions above, we review the evaluation criteria proposed in Chapter 1. For the first objective, we have completely fulfilled it with our specification refinement and synthesis mechanisms. For the second objective, we have developed an initial solution where a certain set of programs with unknown components can now be verified against our interested shape and pure properties, whereas there is still some space for improvement to enlarge the set of programs that can be verified. For the third objective, we have successfully proven the soundness and basic feasibility of our designed theories with the experiments; an ongoing work is to test our verifier with larger-sized systems for its scalability. To conclude, we have accomplished the core part of the requirements stated in the criteria in Chapter 1, and especially all the requirements for the first object. We will discuss possible improvement to this thesis in the next chapter.

7.3 Summary

This chapter has reported the experimental results obtained from our implemented systems, where the results exhibit the soundness and feasibility of our approach. Based on the results we investigated our achievement in terms of the previously proposed evaluation criteria, with the conclusion that our contributions have generally met the requirement of those criteria.

Chapter 8

Conclusion

The goal of this thesis is to build a verification system for both memory safety and functional correctness of programs manipulating pointer-based data structures, which can deal with two scenarios where only partial information about the program is available. In one of the scenarios, the program is annotated only with partial specifications; in the other scenario the program's code is partially available due to invocations to unknown components. This thesis aims to solve these two problems of program verification.

8.1 Main Results

To handle the two scenarios stated above, this thesis clearly defines a programming language as the target to be verified, which is simple yet has the essential properties of a practical programming language that can manipulate heap-based data structures. The operational semantics of such language is also declared. To capture various levels of program correctness, we exploit a specification language based on

separation logic, which allows users to define their own predicates to specify the program properties that they want to verify. This language is quite expressive as it covers aspects from shape to numerical and from relational to content. Its semantic model is introduced as well. Such programming language and specification language serve as the foundation of this thesis' work.

Based on the two languages, this thesis develops solutions to the aforesaid two scenarios. For the first scenario, we propose a new approach to program verification allowing users to provide only partial specification with shape information to methods. Our approach will then refine the given annotations into more complete specifications by discovering missing constraints. The discovered constraints may involve both quantitative and multiset properties that could be later confirmed or revised by users. On the basis of this result, we further augment our approach by requiring only partial specification to be given for primary methods of a program. Specifications for loops and auxiliary methods can then be systematically discovered by our augmented mechanism, with the help of information propagated from the primary methods. This work is aimed at verifying beyond shape properties, with the eventual goal of analysing both memory safety and functional properties for pointer-based data structures.

For the second scenario, we have proposed a top-down verification framework to deal with the verification of such heap-manipulating programs with invocations to unknown components. Provided with a Hoare-style specification $\{\Phi_{pr}\}$ prog $\{\Phi_{po}\}$ where program prog contains calls to some unknown procedure unknown, we infer a specification $mspec_u$ for unknown from the calling contexts, such that the problem of verifying prog can be safely reduced to the problem of proving that the procedure unknown (once its code is available) meets the derived specification $mspec_u$. The expected specification $mspec_u$ is automatically calculated using an abduction-based shape analysis specifically designed for a combined abstract domain.

As a proof of both theories, we have implemented systems to validate their viability. It has been confirmed with experiments that, in the first scenario, we can automatically refine partial specifications with non-trivial constraints, thus making it easier for users to handle specifications with richer properties; in the second scenario, we can verify a considerable set of programs with unknown components with specifications of such unknown components discovered. With comparisons between preset evaluation criteria and our achievements, we draw the conclusion that our objectives are generally met to a large extent. Meanwhile, there are also some aspects of the work that can be further improved, which are introduced in the following section.

8.2 Future Works

In this section, we propose some suggestions on the limitations of this thesis and how to improve them further.

8.2.1 Arrays

In this thesis, we address verification of programs manipulating pointer-based linked data structures but not general "heap-manipulating programs", because our current work cannot handle programs which make use of *arrays*. So far, we have employed reasoning tools for reachability and its relevant properties, described in the form of predicates in our specification language. To handle arrays and pointer arithmetics, we will have to exploit some other techniques, such as Calcagno et al. (2006) and Gulwani et al. (2008). The work (Calcagno et al., 2006) is founded on separation logic, where some restrictions are added over pointer arithmetics such that they are under control of the verification. Gulwani et al. (2008) constructs a lifted

abstract domain which is capable of representing universally quantified facts such as "$\forall i \cdot (0 \leq i < n) \rightarrow a[i]=0$". It might be possible to incorporate these techniques to arrays in our verifier in order to verify a wider class of heap-manipulating programs.

8.2.2 Automation Level

We verify programs with partial specifications by refining such specifications and dividing procedures into primary and auxiliary ones. Its aim is to increase the automation level of the verification and to release users from writing complicated annotations. Following this path, one potential future work is to raise its automation further with more in-depth program analysis techniques. We have investigated this facet as a loop invariant inference (Luo et al., 2010b; Qin et al., 2010). Different from the work reported in this thesis, it conducts a fixed-point iteration process directly over the combined abstract domain, equipped with newly designed abstraction mechanism, and join and widening operators. Using these techniques and the abduction defined in Chapter 6, we intend to build a verification system similar as Calcagno et al. (2009), such that users do not need to annotate anything; instead all the possible specifications of a program will be computed automatically for users to choose from. This approach can be regarded as "fully automatic"; however a significant problem that we can foresee is its scalability —- as our domain is far more sophisticated than that of Calcagno et al. (2009), our analysis will be much more expensive than theirs accordingly. We envisage that this analysis should require some heuristics for better performance, or users may provide some hint for the analysis to conduct abstraction. We have now achieved some results Luo et al. (2010b); Qin et al. (2010) whereas the work is still ongoing.

8.2.3 User Interaction

We already allow users to interact with our verification system by supplying annotations as they want to. However, it is still possible to enable users to interact with the system in other forms, such as dynamic program execution, or rather testing. As aforementioned in Chapter 2, static program verification and analysis generally provides more coverage of program execution paths as well as a proof of absence of bugs; yet it is generally more expensive and less accessible to end users, whereas testing is simpler and more broadly applied in software quality assurance processes. Therefore if there are ways to combine them for our verifier such that the verification result can be improved, then it will be a real advantage. In this proposed aspect of improvement, we want to allow users to provide either static constraints as refinement (in which our analysis may degenerate down to a "pure" verification) or testing data to discharge some of the complexity of the verification to run-time checks. This idea shares some similarity with Gupta et al. (2009), where the authors strengthen static constraint generation with information from both static abstract interpretation (analysis) and dynamic execution of the program (testing). Such strengthening brings in additional linear constraints that may simplify the solver's work and make constraint solving more scalable. If we can successfully apply such idea to our combined domain for shape, numerical and multiset properties, then it should benefit greatly the whole verification process.

8.2.4 Sequential Invocations to Unknown Components

The most challenging part of our verification for partially available programs is the verification of sequential invocation to unknown components. Our current solution tries to find a common specification (*requires* Φ_{pr} *ensures* Φ_{po}) suitable for all the

unknown procedures. Generally we may allow the unknown procedures to have different specifications, possibly with a more in-depth analysis which examines the available code fragments in between those unknown calls. In the case of two unknown calls $unk_0(\boldsymbol{x_0}; \boldsymbol{y_0}); e_1; unk_1(\boldsymbol{x_1}; \boldsymbol{y_1})$, the precondition for unk_0 and the postcondition for unk_1 can be obtained as usual (by analysing the code before unk_0 and after unk_1 respectively). To derive the postcondition Φ^0_{po} for unk_0 and the precondition Φ^1_{pr} for unk_1, we initialise Φ^0_{po} to be `emp` to start a forward analysis over e_1 with abduction, to accumulate (via abduction) the expected behaviour of unk_0 (for e_1 to be verified) as Φ^0_{po}, and extract a formula (which is relevant to the footprint of unk_1) from the abstract state at the end of e_1 as Φ^1_{pr}. However, our initial experiments show that, unless the fragment e_1 is sufficiently complex to expose enough information expected from unk_0, the derived Φ^0_{po} and Φ^1_{pr} can be rather weak. As a consequence, the derived specification for unk_0 can be too weak (with a weak postcondition) and the one for unk_1 can be too strong (with a weak precondition). It remains an open problem how we might tune the derived results to obtain more reasonable specifications. We conjecture that certain heuristics might help and we will explore this further in our future work.

8.3 Summary

This chapter summarises the whole thesis with its achieved results and potential ways of improvement. The results include the solutions to the verification of heap-manipulating programs with partial specifications and partial code, respectively. The improvement concerns the following facets: an enlarged set of programs with arrays to verify, more automation level, improved user interaction, and the verification of programs with sequential invocations to unknown components. These facets depict a roadmap for future works of this thesis.

Bibliography

Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2002. ISBN 1-58113-450-9.

Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis by predicate abstraction. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2005.

Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking Software*, 2001.

Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. *ACM SIGPLAN Notices*, 36:203–213, 2001.

Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3:27–56, 2004a.

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2004b.

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M.

Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, 2006.

Richard Barry. *FreeRTOS Reference Manual - API Functions and Configuration Options*. Online published, 2006.

Boris Beizer. *Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., 1996.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, 2004.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems*, 2005a.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, 2005b.

Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *International Conference on Computer Aided Verification*, 2006.

Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *International Conference on Computer Aided Verification*, 2007.

Richard Bornat. Proving pointer programs in hoare logic. In *International Conference on Mathematics of Program Construction*, 2000.

Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, Ahmed Rezine, and Mihaela Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In *International Conference on Computer Aided Verification*, 2010.

Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. *International Journal on Software Tools for Technlogy Transfer*, 7:212–232, 2005.

Rod M. Burstall. Program proving as hand simulation with a little induction. In *International Federation for Information Processing Congress*, pages 308–312, 1974.

Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2245, pages 108–119, 2001.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *International Static Analysis Symposium*, pages 182–203, 2006.

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Annual IEEE Symposium on Logic in Computer Science*, 2007.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2009.

Néstor Cataño and Marieke Huisman. Chase: A static checker for jml's assignable clause. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2003.

Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2008.

BIBLIOGRAPHY

Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *International Static Analysis Symposium*, 2007.

Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1999.

Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. A reachability predicate for analyzing low-level software. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2007.

Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Department of Computer Science, Iowa State University, 2003.

Yoonsik Cheon and Gary T. Leavens. A quick overview of larch/c++. *Journal of Object-Oriented Programming*, 7:39–49, 1994.

Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties. In *IEEE International Conference on Engineering of Complex Computer Systems*, 2007.

Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular oo verification with separation logic. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2008.

Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, In press:N/A, 2010.

Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1993.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. In *International Conference on Computer Aided Verification*, 2000.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of ACM*, 50:752–794, 2003.

Edmund M. Clarke. Programming language constructs for which it is impossible to obtain good hoare-like axioms. *Journal of ACM*, 26:129–147, 1979.

Edmund M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Workshop of Logic of Programs*, volume 131, 1981.

David R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4:77–103, 2005.

David R. Cok and Joseph R. Kiniry. Esc/java2: Uniting esc/java and jml: Progress and issues in building and using esc/java2, including a case study involving the use of the tool to verify portions of an internet voting tally system. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2004.

Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *European Symposium on Programming*, 2007.

Byron Cook, Ashutosh Gupta, Stephen Magill, Andrey Rybalchenko, Jiri Simsa, Satnam Singh, and Viktor Vafeiadis. Finding heap-bounds for hardware synthesis. In *International Conference on Formal Methods in Computer-Aided Design*, 2009.

Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7:70–90, 1978.

Patrick Cousot. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *International Symposium on Programming*, pages 106–130, 1976.

Patrick Cousot and Radhia Cousot. *Logics and Languages for Reliability and Security*, chapter N/A. IOS Press, 2010.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, 1977.

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1979.

Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1978.

Rick D. Craig and Stefan P. Jaskiel. *Systematic Software Testing*. Artech House, Inc., 2002.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of ACM*, 52:365–473, 2005.

Edsger W. Dijkstra. *A discipline of programming.* Prentice-Hall, 1976.

Dino Distefano. Attacking large industrial code with bi-abductive inference. In *International Workshop on Formal Methods for Industrial Critical Systems*, 2009.

Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2006.

Lucas Dixon. *A Proof Planning Framework for Isabelle.* PhD thesis, University of Edinburgh, 2005.

Maryam Emami. A practical inter-procedural alias analysis for an optimizing/paralleling c compiler. Master's thesis, School of Computer Science, McGill University, 1993.

Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1994.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics (Mathematical Aspects of Computer Science)*, 1967.

Roberto Giacobazzi. Abductive analysis of modular logic programs. In *International Symposium on Logic Programming*, 1994.

Patrice Godefroid. Model checking for programming languages using verisoft. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1997.

Denis Gopan and Thomas Reps. Low-level library analysis and summarization. In *International Conference on Computer Aided Verification*, 2007.

Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *International Static Analysis Symposium*, 2006.

Sumit Gulwani and Ashish Tiwari. Computing procedure summaries for interprocedural analysis. In *European Symposium on Programming*, 2007.

Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2008.

Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. A combination framework for tracking partition sizes. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 239–251, 2009.

Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. From tests to proofs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In *Programs as Data Objects II*, 2001.

Peter Habermehl, Radu Iosif, and Tomáš Vojnar. Automata-based verification of programs with tree updates. *Acta Informatica*, 47:1–31, 2010.

Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2005.

Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1995.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with BLAST. In *SPIN Workshop on Model Checking Software*, 2003.

William C. Hetzel and Bill Hetzel. *The Complete Guide to Software Testing*. John Wiley & Sons, Inc., 1991.

Charles A. R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12:576–583, 1969.

Charles A. R. Hoare and Jifeng He. A trace model for pointers and objects. In *European Conference on Object-Oriented Programming*, 1999.

Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

Information Management and Technology Division. Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia. Technical report, General Accounting Office, US, 1992.

Andrew Ireland. A cooperative approach to loop invariant discovery for pointer programs. In *Workshop on Invariant Generation*, 2007.

Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2001.

Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference*, 2002.

Cliff Jones, Peter O'Hearn, and Jim Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39:93–95, April 2006.

Cem Kaner. Exploratory testing after 23 years. In *Conference of the Association for Software Testing*, 2006.

Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, 2001.

Wojtek Kozaczynski and Grady Booch. Component-based software engineering. *IEEE Software*, 15:34–36, 1998.

Viktor Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, 2007.

Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2002.

Viktor Kuncak, Patrick Lam, Karen Zee, and Martin C. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32:988–1005, 2006.

Shuvendu K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2006.

Patrick Lam. *The Hob System for Verifying Software Design Properties*. PhD thesis, Massachusetts Institute of Technology, 2007.

William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In *International Symposium on Formal Methods*, 1999.

Gary T. Leavens, Clyde Ruby, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *International Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2000.

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31:1–38, 2006.

Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *European Symposium on Programming*, 2005.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conferences on Logic for Programming, Artificial Intelligence and Reasoning*, 2010.

Nancy G. Leveson and Clark S. Turner. Investigation of the therac-25 accidents. *IEEE Computer*, 26:18–41, 1993.

Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying crosscutting features as open systems. *ACM SIGSOFT Software Engineering Notes*, 27:89–98, 2002.

Jacques-Louis Lions. Ariane 5 - flight 501 failure - report by the inquiry board. Technical report, Académie des Sciences, France, 1996.

Richard J. Lipton. A necessary and sufficient condition for the existence of hoare logics. In *IEEE Symposium on Foundations of Computer Science*, 1977.

Chenguang Luo, Florin Craciun, Shengchao Qin, Guanhua He, and Wei-Ngan Chin. Verifying pointer safety for programs with unknown calls. *Journal of Symbolic Computation*, 45:1163–1183, 2010a.

Chenguang Luo, Guanhua He, Shengchao Qin, and Wei-Ngan Chin. Loop invariant synthesis in a combined domain (extended abstract). In *Verified Software: Theory, Tools and Experiments*, 2010b.

Ewen Maclean, Andrew Ireland, Robert Atkey, and Lucas Dixon. Refinement and term synthesis in loop invariant generation. In *Workshop of Invariant Generation*, 2009.

Stephen Magill, Josh Berdine, Edmund M. Clarke, and Byron Cook. Arithmetic strengthening for shape analysis. In *International Static Analysis Symposium*, 2007.

Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Thor: A tool for reasoning about shape and arithmetic. In *International Conference on Computer Aided Verification*, 2008.

Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2010.

Roman Manevich. *Partially Disjunctive Shape Analysis*. PhD thesis, Tel-Aviv University, 2009.

Zohar Manna and Amir Pnueli. Axiomatic approach to total correctness of programs. *Acta Informatica*, 3:243–264, 1974.

Ken L. McMillan. *Symbolic model checking —- an approach to the state explosion problem*. PhD thesis, SCS, Carnegie Mellon University, 1992.

Anders Möller and Michael I. Schwartzbach. The pointer assertion logic engine. *ACM SIGPLAN Notices*, 36:221–231, 2001.

Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 330–341, 2004.

Madanlal S. Musuvathi, David Park, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *Symposium on Operating Systems Design and Implementation*, 2002.

Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1:245–257, 1979.

Huu Hai Nguyen and Wei-Ngan Chin. Enhancing program verification with lemmas. In *International Conference on Computer Aided Verification*, 2008.

Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2007.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.

Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5:215–244, 1999.

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*, 2001.

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2004.

Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2008.

Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In *IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification*, 1999.

Alexandre Petit-Bianco. Java garbage collection for real-time systems. *Dr Dobb's Journal of Software Tools and Professional Programmers*, 23:8, 1998.

Erik Poll, Joachim van den Berg, and Bart Jacobs. Specification of the javacard api in jml. In *Working Conference on Smart Card Research and Advanced Applications*, 2001.

Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *Asian Computing Science Conference*, 2006.

Shengchao Qin, Guanhua He, Chenguang Luo, and Wei-Ngan Chin. Loop invariant synthesis in a combined domain. In *International Conference on Formal Engineering Methods*, 2010.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1995.

Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.

John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Annual IEEE Symposium on Logic in Computer Science*, 2002.

John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*, 1999.

John C. Reynolds. An overview of separation logic. In *Verified Software: Theory, Tools and Experiments*, 2005.

Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. *ACM SIGPLAN Notices*, 40:296–309, 2005.

Manuel Rubio-Sánchez, Jaime Urquiza-Fuentes, and Cristóbal Pareja-Flores. A gentle introduction to mutual recursion. In *Annual Conference on Innovation and Technology in Computer Science Education*, 2008.

Radu Rugina. Quantitative shape analysis. In *International Static Analysis Symposium*, 2004.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1999.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24: 217–298, 2002.

Peter Schmitt, Isabel Tonin, Claus Wonnemann, Eric Jenn, Stéphane Leriche, and James J. Hunt. A case study of specification and verification using jml in an avionics application. In *International Workshop on Java Technologies for Real-time and Embedded Systems*, 2006.

Roger Sessions. *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley & Sons, Inc., 1998.

Arthur G. Stephenson, Daniel R. Mulville, Frank H. Bauer, Greg A. Dukeman, Peter Norvig, Edward J. Weiler, Lia S. LaPiana, Peter J. Rutledge, David Folta, and

Robert Sackheim. Mars climate orbiter mishap investigation board phase i report. Technical report, NASA, US, 1999.

Clemens Szyperski. Component technology: what, where, and how? In *International Conference on Software Engineering*, 2003.

Mana Taghdiri. *Automating Modular Verification by Refining Specifications*. PhD thesis, EECS Department, Massachusetts Institute of Technology, 2008.

Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill Professional, 1999.

Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering Journal*, 10:3–12, 2003.

Thomas Wies, Viktor Kuncak, Karen Zee, Andreas Podelski, and Martin Rinard. On verifying complex properties using symbolic shape analysis. In *The Computing Research Repository*, 2006.

Thomas Wies, Viktor Kuncak, Karen Zee, Andreas Podelski, and Martin Rinard. On verifying complex properties using symbolic shape analysis. In *Workshop of Heap Analysis and Verification*, 2007.

Jim Woodcock. Verified software grand challenge. In *International Symposium on Formal Methods*, 2006.

Gaoyan Xie and Zhe Dang. Testing systems of concurrent black-boxes —- an automata-theoretic and decompositional approach. In *International Workshop on Formal Approaches to Software Testing*, 2005.

Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In *International Conference on Foundations of Software Science and Computation Structures*, 2002.

Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *International Conference on Computer Aided Verification*, 2008.

Greta Yorsh, Er Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. In *International Conference on Foundations of Software Science and Computation Structures*, 2006.

Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. *ACM SIGPLAN Notices*, 43:221–234, 2008.

BIBLIOGRAPHY

Appendix A

Soundness Proofs

A.1 Soundness of Specification Refinement

We have defined the underlying operational semantics of our language in Chapter 3. Its concrete program state consists of stack s and heap h. We have also defined the relation $s, h \models \Delta$ and the transition $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', \nu\rangle$. Before proceeding to the soundness definition, recalling that we have both unprimed variables (for their initial values in abstract states) and primed ones (for their current values), we realise that the concrete program states should always be linked to the primed ones. For this reason we have the following definition:

Definition A.1.1 (Poststate) *Given an abstract state Δ, Post(Δ) captures the relation between primed variables of Δ. That is,*

$$Post(\Delta) =_{df} \rho(\exists V \cdot \Delta), \text{ where}$$
$$V = \{v_1, \ldots, v_n\} \text{ denotes all unprimed program variables in } \Delta, \text{ and}$$
$$\rho = [v_1/v_1', \ldots, v_n/v_n'].$$

For example, for $\Delta = \mathtt{x'}{::}\mathtt{Node}\langle \mathtt{v'}, \mathtt{y'}\rangle \wedge \mathtt{v'}{=}\mathtt{v} \wedge \mathtt{y'}{=}\mathtt{null}$, *we have* $Post(\Delta) =$

$\texttt{x::Node}\langle \texttt{v}, \texttt{y}\rangle \wedge \texttt{y=null}$.

Then we define the soundness of our refinement as follows:

Definition A.1.2 (Soundness) *For a method definition* $t\ mn\ ((\boldsymbol{t\ u}); (\boldsymbol{t\ v}))\ \{e\}$, *if our verification refines its specification as* $t\ mn\ ((\boldsymbol{t\ u}); (\boldsymbol{t\ v}))$ *requires* Φ_{pr} *ensures* Φ_{po} $\{e\}$, *then for all* $s, h \models Post(\Phi_{pr})$, *if* $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', \text{-}\rangle$, *then we have* $s', h' \models Post(\Phi_{po})$.

The soundness of our verification is ensured by the soundness of the following: the entailment prover, the pure fixed-point calculation, the pure abstraction derivation, and the abstract semantics (w.r.t. the concrete semantics). Among the above, the soundness of the entailment prover and pure fixed-point calculation are already confirmed (Chin et al., 2010; Nipkow et al., 2002; Popeea and Chin, 2006), and hence we will concentrate on the soundness of abstract semantics and pure abstraction derivation.

Lemma A.1.3 (Sound pure abduction) *If* $\sigma_1 \wedge [\sigma'] \rhd \sigma_2 * \sigma_3$, *then* $\forall s, h \models Post(\sigma_1 \wedge \sigma')$, *we have* $s, h \models Post(\sigma_2 * \sigma_3)$.

Proof This is ensured by the entailment relationship in the premise of each of the pure abduction rules and the soundness of the entailment checking (Chin et al., 2010). □

Lemma A.1.4 (Sound abstract semantics) *If* $[\![e]\!]_{\mathcal{I}}(\Delta, 0) = (\Delta_1, 0)$, *then for all* s, h, *if* $s, h \models Post(\Delta)$ *and* $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$, *then there always exists* Δ_0 *such that*

$$s_1, h_1 \models Post(\Delta_0) \qquad \textit{and} \qquad [\![e_1]\!]_{\mathcal{I}}(\Delta_0, 0) = (\Delta_1, 0)$$

Proof The proof is done by structural induction over program constructors:

- Case $\texttt{null} \mid k \mid v \mid v.f$. Straightforward.

- Case $v = e$. There are two cases according to the operational semantics:

  - e is not a value. From operational semantics, there is e_1 s.t. $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$, and $\langle s, h, v{=}e\rangle \hookrightarrow \langle s_1, h_1, v{=}e_1\rangle$. From abstract semantics for assignment, if $[\![e]\!]_{\mathcal{T}}(\Delta, 0) = (\Delta_2, 0)$, and $\Delta_1{=}[v_1/v', r_1/\texttt{res}](\Delta_2) \wedge v'{=}r_1$. By induction hypothesis, there exists Δ_0, $s_1, h_1 \models \Delta_0$ and $[\![e_1]\!]_{\mathcal{T}}(\Delta_0, 0) = (\Delta_2, 0)$. It concludes from the assignment rule that $[\![v = e_1]\!]_{\mathcal{T}}(\Delta_0, 0) = (\Delta_1, 0)$.

  - e is a value. Trivial.

- Case $\texttt{new}\ c(\boldsymbol{v})$. From abstract semantics for $\texttt{new}$, we have $[\![\texttt{new}\ c(\boldsymbol{v})]\!]_{\mathcal{T}}(\Delta, 0) = (\Delta_1, 0)$, where $\Delta_1 = \Delta * \texttt{res}{::}c\langle v'_1, \ldots, v'_n\rangle$. Let $\Delta_0 = \Delta_1$. From the operational semantics, we have $\langle s, h, \texttt{new}\ c(\boldsymbol{v})\rangle \hookrightarrow \langle s, h{+}[\iota \mapsto r], \iota\rangle$, where $\iota \notin dom(h)$. From $s, h \models \Delta$, we have $s, h{+}[\iota \mapsto r] \models \Delta_0$. Moreover, $[\![\iota]\!]_{\mathcal{T}}(\Delta_0, 0) = (\Delta_1, 0)$.

- Case $v_1.f = v_2$. Take $\Delta_0 = \Delta$. It concludes immediately from the exec rule for field update and the underlying operational semantics.

- Case $\texttt{free}(x)$. Denote Δ as $\bigvee_i (x{::}c\langle \boldsymbol{y}_i\rangle * \sigma_i)$ and Δ_0 as $\bigvee_i \sigma_i$, then from $\texttt{free}$'s operational semantics we know that if $s, h \models \mathit{Post}(\Delta)$ and $\langle s, h, \texttt{free}(x)\rangle \hookrightarrow \langle s_1, h_1, \text{-}\rangle$, then $s_1, h_1 \models \mathit{Post}(\Delta_0)$ and $\Delta_0 = \Delta_1$.

- Case $e_1; e_2$. We consider the case where e_1 is not a value (otherwise it is trivial). From the operational semantics, we have $\langle s, h, e_1\rangle \hookrightarrow \langle s_1, h_1, e_3\rangle$. From the abstract semantics rule for sequence, we have $\vdash \{\Delta\} e_1 \{\Delta_2\}$. By induction hypothesis, there exists Δ_0 s.t. $s_1, h_1 \models \mathit{Post}(\Delta_0)$, and $\vdash \{\Delta_0\} e_3 \{\Delta_2\}$. By the sequential rule we have $[\![e_3; e_2]\!]_{\mathcal{T}}(\Delta_0, 0) = (\Delta_1, 0)$.

- Case $\texttt{if}\ (v)\ e_1\ \texttt{else}\ e_2$. There are two possibilities in the operational semantics:

– $s(v)$=`true`. We have $\langle s, h, \texttt{if}\ (v)\ e_1\ \texttt{else}\ e_2\rangle \hookrightarrow \langle s, h, e_1\rangle$. Let Δ_0=($\Delta \wedge v'$). It is obvious that $s, h \models \Delta_0$. From the if-conditional rule of abstract semantics, we have:

$$[\![e_1]\!]_{\mathcal{T}}(\Delta_0, 0) = (\Delta_2, 0)$$
$$[\![e_2]\!]_{\mathcal{T}}(\Delta \wedge \neg v', 0) = (\Delta_3, 0)$$

And we also have (due to sound weakening of postcondition)

$$[\![e_1]\!]_{\mathcal{T}}(\Delta_0, 0) = (\Delta_2 \vee \Delta_3, 0)$$

That is, $[\![e_1]\!]_{\mathcal{T}}(\Delta_0, 0) = (\Delta_1, 0)$.

– $s(v) =$ `false`. Analogous.

- Case $mn(v_{1\ldots n})$. For the method invocation rule, we know $\Delta \vdash [v'_j/v_j]_{j=1}^{n} \Phi^i_{pr} * \Delta^i$, for $i = 1, \ldots, p$. Take $\Delta_0 = \bigvee_{i=1}^{p} [v'_j/v_j]_{j=1}^{n} \Phi^i_{pr} * \Delta^i$. From the operational semantics and the above heap entailment, we have $s_1, h_1 \models \Delta_0$. Then the method invocation rule implies $\forall i \in 1 \ldots p \cdot [\![e_1]\!]_{\mathcal{T}}([v'_j/v_j]_{j=1}^{n} \Phi^i_{pr} * \Delta^i, 0) = (\Delta^i * \Phi^i_{po}, 0)$. Therefore we have $[\![e_1]\!]_{\mathcal{T}}(\Delta_0, 0) = (\Delta_1, 0)$ which concludes.

- Case `while` (v) $\{e\}$. It can be converted to tail-recursive method call with all parameters passed by reference, and thus follows the above case. □

Lemma A.1.5 (Sound pure constraint abstraction) *Given a method with pre/post shape templates* `pre` *and* `post`*, if our analysis successfully computes a constraint abstraction* `Q` *in the first step without abduction, and derives a pure constraint* `P` *in the second step, then we have* $\texttt{Q} \vdash \texttt{post} \wedge \texttt{P}$.

Proof This proof follows directly our procedure to compute the pure constraint abstraction from the shape one, plus the soundness of entailment checking and abduction. Denote the shape constraint abstraction as

$$\texttt{Q} ::= \bigvee_i \texttt{Q}_i$$

and the provided post-shape `post`, we use

$$\texttt{Q}_i \vdash \texttt{post} \wedge \texttt{P}_i$$

to derive each $\mathtt{P}_i$, and construct

$$\mathtt{P} ::= \bigvee_i \mathtt{P}_i$$

Therefore, our result $\mathtt{Q} \vdash \mathtt{post} \wedge \mathtt{P}$ can be obtained from the fact that $\bigvee_i \mathtt{Q}_i \vdash \mathtt{post} \wedge (\bigvee_i \mathtt{P}_i)$. □

Then based on the discussion above we have:

Theorem A.1.6 (Soundness) *Our verification is sound with respect to the underlying operational semantics.*

We have one more note about the post verification conducted in line 12 of our main algorithm in Figure 4.4. Such verification is used to confirm that the strengthened precondition can guarantee memory safety. This added precaution is because sometimes our refinement of precondition might not be sufficient for the program to execute without inappropriate memory access, which could be attributed to the fact that users have not provided a good enough predicate to describe the obligation of memory safety. For example, if our verification is only supplied with a list predicate which does not contain its length information, then we can never obtain the prerequisite "the input length should be at least `n`", even if the memory safety requires that. Hence our post verification will rule out this case. However it does not affect the soundness of our verification: the memory violation will incur `false` as an abstract state which implies any postcondition we may discover. The only reason we need it is as aforesaid —- to leave only meaningful refined specifications (which has safety guarantee for the program) in our results.

A.2 Soundness of Shape Specification Synthesis

The soundness definition of shape specification synthesis follows the one in the previous section, referring to the underlying operational semantics:

Definition A.2.1 (Soundness) *For a method definition* $t\ mn\ ((\boldsymbol{t}\ \boldsymbol{u});(\boldsymbol{t}\ \boldsymbol{v}))\ \{e\}$, *if our analysis refines its specification as* $t\ mn\ ((\boldsymbol{t}\ \boldsymbol{u});(\boldsymbol{t}\ \boldsymbol{v}))$ *requires* Φ_{pr} *ensures* $\Phi_{po}\ \{e\}$, *then for all* $s, h \models Post(\Phi_{pr})$, *if* $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', \text{-}\rangle$, *then we have* $s', h' \models Post(\Phi_{po})$.

Therefore the soundness of the whole approach can be reduced to the soundness of our synthesis of shape specifications. To prove this, we need to review our precondition/postcondition synthesis algorithms. From these two algorithms, we can see that our synthesised pre-shape must satisfy the abstract state at the calling context (because of the entailment relationship), and the post-shape is checked to see whether it could possibly be abstracted as the execution result of the unfolded program. From the soundness of entailment checking and abduction, we have

Theorem A.2.2 (Soundness) *Our verification is still sound with respect to the underlying operational semantics, with the specification synthesis mechanism added.*

Proof The soundness is proven with the following claims:

- For pre-shape synthesis, we can see in line 3 of Figure 5.6 that for each chosen shape candidate σ_C, we always have $\sigma \vdash [\boldsymbol{x}/\boldsymbol{u}, \boldsymbol{y}/\boldsymbol{v}]\sigma_C$ where σ is the abstract state in the calling context.
- For post-shape synthesis, line 5 of Figure 5.7 suggests that the chosen shape candidate σ_C has the potential to establish the expected post-state Δ (the symbolic execution result of the unfolded program), possibly with some additional

pure constraints σ (which are to be discovered by the following refinement process, should the refinement succeed).

- Finally, we already know that the entailment checking, the pure abduction, the symbolic execution and the refinement are all sound with respect to the underlying semantics. This concludes the proof. □

A.3 Soundness of Verification for Programs with Unknown Components

Informally, in the presence of invocations to unknown components, the soundness of the verification signifies that, a program is successfully verified against its specifications, if all the unknown procedures that it invokes conform to the specifications discovered by the verification algorithm. Therefore, the correctness of the program depends on a (possible) further verification for the unknown procedures. It can be defined as follows:

Definition A.3.1 (Soundness) *Suppose that for specification table $\mathcal{T}$, program to be verified $v = \{e_1; u; e_2\}$ and its specifications $mspec_v$, our verification succeeds and returns $\mathcal{T}_u$ as the specification table for unknown procedures invoked in v. Then we say our verification is sound, if the following holds:*

$$\forall \sigma \in [\![e_1; u; e_2]\!]_{\mathcal{T} \uplus \mathcal{T}_U} \{[\boldsymbol{x_0}/\boldsymbol{a}, \boldsymbol{y_0}/\boldsymbol{b}]\Phi_{pr}\} \cdot \sigma \vdash [\boldsymbol{x_0}/\boldsymbol{a}, \boldsymbol{y_0}/\boldsymbol{b}, \boldsymbol{y'_0}/\boldsymbol{b'}]\Phi_{po} * \texttt{true}$$

which means that, with respect to the underlying semantics, if all the unknown procedures can be verified to satisfy their specifications in $\mathcal{T}_u$, then the whole program v should meet all the specifications in $mspec_v$.

To prove that our verification is sound, we proceed with the soundness of three aspects, that is, entailment checking, abduction and abstract semantics for forward analysis. The soundness of entailment checking is already stated in Section A.1. For the soundness of abduction, we have

Lemma A.3.2 (Sound abduction) *If* $\sigma_1 * [\sigma'] \rhd \sigma_2 * \sigma_3$, *then* $\forall s, h \models Post(\sigma_1 * \sigma')$, *we have* $s, h \models Post(\sigma_2 * \sigma_3)$.

Proof This is ensured by the entailment relationship in the premise of each of the first three abduction rules and the soundness of the entailment checking (Chin et al., 2010). For the last rule, it is sound as well because $\sigma * \sigma_1 \vdash \sigma_1 * \texttt{true}$. □

Lemma A.3.3 (Sound underlying abstract semantics) *If* $[\![e]\!]_{\mathcal{T}}\Delta = \Delta_1$, *then for all* s, h, *if* $s, h \models Post(\Delta)$ *and* $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$, *then there always exists* Δ_0 *such that*

$$s_1, h_1 \models Post(\Delta_0) \qquad \textit{and} \qquad [\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$$

Proof The proof is done by structural induction over program constructors:

- Case $\texttt{null} \mid k \mid v \mid v.f$. Directly obtained from definition.
- Case $\texttt{new}\ c(\boldsymbol{v})$. From abstract semantics for $\texttt{new}$, we have $[\![\texttt{new}\ c(\boldsymbol{v})]\!]_{\mathcal{T}}\Delta = \Delta_1$, where $\Delta_1 = \Delta * \texttt{res}{::}c\langle v'_1, \ldots, v'_n\rangle$. Let $\Delta_0 = \Delta_1$. From the operational semantics, we have $\langle s, h, \texttt{new}\ c(\boldsymbol{v})\rangle \hookrightarrow \langle s, h{+}[\iota \mapsto r], \iota\rangle$, where $\iota \notin dom(h)$. From $s, h \models \Delta$, we have $s, h{+}[\iota \mapsto r] \models \Delta_0$. Moreover, $[\![\iota]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$.
- Case $\texttt{free}(x)$. Denote Δ as $\bigvee_i (x{::}c\langle \boldsymbol{y}_i\rangle * \sigma_i)$ and Δ_0 as $\bigvee_i \sigma_i$, then from $\texttt{free}$'s operational semantics we know that if $s, h \models Post(\Delta)$ and $\langle s, h, \texttt{free}(x)\rangle \hookrightarrow \langle s_1, h_1, \text{-}\rangle$, then $s_1, h_1 \models Post(\Delta_0)$ and $\Delta_0 = \Delta_1$.
- Case $v_1.f = v_2$. Take $\Delta_0 = \Delta$. It concludes immediately from the exec rule for field update and the underlying operational semantics.

- Case $v = e$. There are two cases according to the operational semantics:
  - e is a value. Straightforward.
  - e is not a value. From the underlying operational semantics, there is e_1 s.t. $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$, and $\langle s, h, v{=}e\rangle \hookrightarrow \langle s_1, h_1, v{=}e_1\rangle$. From abstract semantics for assignment, if we have $\Delta_1{=}[v_1/v', r_1/\texttt{res}](\Delta_2){\wedge}v'{=}r_1$ where $[\![e]\!]_{\mathcal{T}}\Delta = \Delta_2$, By induction hypothesis, there exists Δ_0, $s_1, h_1 \models \Delta_0$ and $[\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_2$. It concludes from the assignment rule that $[\![v = e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$.

- Case $e_1; e_2$. We consider the case where e_1 is not a value (otherwise it is straightforward). From the operational semantics, we have $\langle s, h, e_1\rangle \hookrightarrow \langle s_1, h_1, e_3\rangle$. From the abstract semantics rule for sequence, we have $\vdash \{\Delta\}e_1\{\Delta_2\}$. By induction hypothesis, there exists Δ_0 s.t. $s_1, h_1 \models \mathit{Post}(\Delta_0)$, and $\vdash \{\Delta_0\}e_3\{\Delta_2\}$. By the sequential rule we have $[\![e_3; e_2]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$.

- Case $\texttt{if}\ (v)\ e_1\ \texttt{else}\ e_2$. There are two possibilities in the operational semantics:
  - $s(v){=}\texttt{true}$. We have $\langle s, h, \texttt{if}\ (v)\ e_1\ \texttt{else}\ e_2\rangle \hookrightarrow \langle s, h, e_1\rangle$. Let $\Delta_0{=}(\Delta{\wedge}v')$. It is obvious that $s, h \models \Delta_0$. From the if-conditional rule of abstract semantics, we have:
    $$[\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_2$$
    $$[\![e_2]\!]_{\mathcal{T}}\Delta{\wedge}\neg v' = \Delta_3$$
    And we also have (due to sound weakening of postcondition)
    $$[\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_2{\vee}\Delta_3$$
    That is, $[\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$.
  - $s(v) = \texttt{false}$. Same.

- Case $mn(v_{1\ldots n})$. For the method invocation rule, we know $\Delta{\vdash}[v'_j/v_j]_{j=1}^{n}\Phi^i_{pr} * \Delta^i$, for $i = 1, \ldots, p$. Take $\Delta_0 = \bigvee_{i=1}^{p}[v'_j/v_j]_{j=1}^{n}\Phi^i_{pr}{*}\Delta^i$. From the operational semantics and the above heap entailment, we have $s_1, h_1 \models \Delta_0$. Then the

method invocation rule implies $\forall i{\in}1\ldots p \cdot [\![e_1]\!]_{\mathcal{T}}[v'_j/v_j]_{j=1}^{n}\Phi^i_{pr}{*}\Delta^i = \Delta^i{*}\Phi^i_{po}$. Therefore we have $[\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$ which concludes.

- Case `while` (v) $\{e\}$. It can be converted to tail-recursive method call with all parameters passed by reference, and thus follows the above case. □

Lemma A.3.4 (Sound abstract semantics with abduction) *If* $[\![e]\!]^{\mathsf{A}}_{\mathcal{T}}(\mathtt{emp}, \mathtt{emp}) = (\Delta_1, \Delta'_1)$*, then for all* s, h*, if* $s, h \models \mathit{Post}(\Delta'_1)$ *and* $\langle s, h, e\rangle{\hookrightarrow}\langle s_1, h_1, e_1\rangle$*, then there always exists* Δ_0 *such that*

$$s_1, h_1 \models \mathit{Post}(\Delta_0) \qquad \textit{and} \qquad [\![e_1]\!]_{\mathcal{T}}\Delta_0 = \Delta_1$$

Proof Generally there are two types of constructors which may alter the result from abduction: heap-sensitive commands $d[x]$ and procedure invocation. We investigate them respectively.

- Case $d[x]$. As we know that $[\![d[x]]\!]^{\mathsf{A}}_{\mathcal{T}}(\mathtt{emp}, \mathtt{emp}) =_{df} \mathsf{Exec}^{\dagger}(d[x])(\mathcal{T}) \circ \mathsf{Unfold}^{\dagger}(x)(\mathtt{emp}, \mathtt{emp})$, we consider the lifted unfolding operation (Unfold) to produce the abduction result. From its definition (Section 6.5.4), since the current abstract state is `emp`, the unfold must fail and `false` $\in \Delta$. Then in the remaining two cases of **if** in its definition, the second one is the trivial case where we conclude with $\Delta_0 = \mathtt{false}$. For the first one, as $x{::}c\langle \boldsymbol{y}\rangle$ is added to both the current state and the abduction result, from the induction assumption proven in Lemma A.3.3 we know that we can find such Δ_0, and the conclusion holds.

- Case $mn(x_1, \ldots, x_m; y_1, \ldots, y_n)(\mathcal{T})(\sigma, \sigma')$. There are two scenarios here: $\sigma \vdash \rho\Phi_{pr} * \sigma_1$ and $\sigma'_1{=}\mathtt{emp}$, or $\sigma * [\sigma'_1] \rhd \rho\Phi_{pr} * \sigma_1$. In the first scenario, the rule degenerates to the case in the underlying semantics. In the second one, Δ'_1 is assigned by the abduction, where the entailment relationship $\Delta'_1 \vdash \rho\Phi_{pr} * \mathtt{true}$ is established. Therefore this case follows the induction assumption from Lemma A.3.3. □

On the basis of above we have

Theorem A.3.5 (Soundness) *Our verification of programs with unknown components is sound.*

Appendix B

Shape Predicates and Program Code Used in Experiments

This chapter presents the definitions of shape predicates and program code used in the experiments which are not introduced in the previous chapters. These predicates and code mainly constitute a proof of theory over the classic algorithms manipulating data structures, and for the sake of length does not include the part of FreeRTOS (Barry, 2006), whose code can be found in its website.

B.1 Shape Predicate Definitions

Below are the definitions of the shape predicates not previously defined:

$$\texttt{dll}\langle \texttt{p}, \texttt{n}\rangle \equiv (\texttt{root}{=}\texttt{p} \wedge \texttt{n}{=}0) \vee (\texttt{root}{::}\texttt{Node2}\langle \texttt{v}, \texttt{p}, \texttt{q}\rangle * \texttt{q}{::}\texttt{dll}\langle \texttt{root}, \texttt{n}_1\rangle \wedge \texttt{n}{=}\texttt{n}_1{+}1)$$

$$\texttt{dllB}\langle \texttt{p}, \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{p} \wedge \texttt{S}{=}\emptyset) \vee (\texttt{root}{::}\texttt{Node2}\langle \texttt{v}, \texttt{p}, \texttt{q}\rangle * \texttt{q}{::}\texttt{dllB}\langle \texttt{root}, \texttt{S}_1\rangle \wedge \texttt{S}{=}\texttt{S}_1 \sqcup \{\texttt{v}\})$$

$$\texttt{sll}\langle \texttt{n}, \texttt{mn}, \texttt{mx}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{n}{=}0 \wedge \texttt{mn}{=}\texttt{mx}) \vee (\texttt{root}{::}\texttt{Node}\langle \texttt{v}, \texttt{q}\rangle * \texttt{q}{::}\texttt{sll}\langle \texttt{n}_1, \texttt{k}, \texttt{mx}\rangle \wedge \texttt{n}{=}\texttt{n}_1{+}1 \wedge \texttt{mn}{\leq}\texttt{k})$$

$$\texttt{sllB2}\langle \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\emptyset) \vee (\texttt{root}{::}\texttt{Node}\langle \texttt{v}, \texttt{q}\rangle * \texttt{q}{::}\texttt{sllB2}\langle \texttt{S}_1\rangle \wedge \texttt{S}{=}\{\texttt{v}\} \sqcup \texttt{S}_1 \wedge (\forall \texttt{x}{\in}\texttt{S}_1 \cdot \texttt{v}{\geq}\texttt{x}))$$

$$\texttt{slsB}\langle \texttt{p}, \texttt{S}\rangle \equiv (\texttt{root}{=}\texttt{p} \wedge \texttt{S}{=}\emptyset) \vee (\texttt{root}{::}\texttt{Node}\langle \texttt{v}, \texttt{q}\rangle * \texttt{q}{::}\texttt{slsB}\langle \texttt{p}, \texttt{S}_1\rangle \wedge \texttt{S}{=}\{\texttt{v}\} \sqcup \texttt{S}_1 \wedge (\forall \texttt{x}{\in}\texttt{S}_1 \cdot \texttt{v}{\leq}\texttt{x}))$$

$$\texttt{bt}\langle \texttt{S}, \texttt{h}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\emptyset \wedge \texttt{h}{=}0) \vee (\texttt{root}{::}\texttt{Node2}\langle \texttt{v}, \texttt{p}, \texttt{q}\rangle * \texttt{p}{::}\texttt{bt}\langle \texttt{S}_\texttt{p}, \texttt{h}_\texttt{p}\rangle * \texttt{q}{::}\texttt{bt}\langle \texttt{S}_\texttt{q}, \texttt{h}_\texttt{q}\rangle \wedge \texttt{S}{=}\texttt{S}_\texttt{p} \sqcup \texttt{S}_\texttt{q} \wedge \texttt{h}{=}1{+}\mathsf{max}(\texttt{h}_\texttt{p}, \texttt{h}_\texttt{q}))$$

$$\texttt{bst}\langle \texttt{sm}, \texttt{lg}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{sm}{=}\texttt{lg}) \vee (\texttt{root}{::}\texttt{Node2}\langle \texttt{v}, \texttt{p}, \texttt{q}\rangle * \texttt{p}{::}\texttt{bst}\langle \texttt{sm}, \texttt{mn}\rangle * \texttt{q}{::}\texttt{bst}\langle \texttt{mx}, \texttt{lg}\rangle \wedge \texttt{mn}{<}\texttt{v}{<}\texttt{mx})$$

$$\texttt{avl}\langle \texttt{S}, \texttt{h}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\emptyset \wedge \texttt{h}{=}0) \vee (\texttt{root}{::}\texttt{Node2}\langle \texttt{v}, \texttt{p}, \texttt{q}\rangle * \texttt{p}{::}\texttt{bt}\langle \texttt{S}_\texttt{p}, \texttt{h}_\texttt{p}\rangle * \texttt{q}{::}\texttt{bt}\langle \texttt{S}_\texttt{q}, \texttt{h}_\texttt{q}\rangle \wedge \texttt{S}{=}\texttt{S}_\texttt{p} \sqcup \texttt{S}_\texttt{q} \wedge \texttt{h}{=}1{+}\mathsf{max}(\texttt{h}_\texttt{p}, \texttt{h}_\texttt{q}) \wedge -1{\leq}\texttt{h}_\texttt{p}{-}\texttt{h}_\texttt{q}{\leq}1)$$

B.2 Program Code

Below is the program code that we have done experiments with, which is not mentioned in the main text:

```
Node2 create2(int n) {
  if (n == 0) return null;
```

```
  else {
    Node2 r = create2(n - 1);
    Node2 s = new Node2(n, null, r);
    if (r != null) r.prev = s;
    return s;
  }
}

void sort_insert(Node x, int v) {
  if (x.next == null)
    x.next = new Node(v, null);
  else if (v <= x.next.val)
    x.next = new Node(v, x.next);
  else
    sort_insert(x.next, v);
}

void sort_insert2(Node2 x, int v) {
  if (x.next == null)
    x.next = new Node2(v, x, null);
  else if (v <= x.next.val) {
    x.next = new Node2(v, x, x.next);
    x.next.next.prev = x.next;
  } else
    sort_insert2(x.next, v);
}

void tail_insert(Node x, int v) {
  if (x.next == null)
```

```
    x.next = new Node(v, null);
  else
    tail_insert(x.next, v);
}

void rand_insert(Node x, int v) {
  int a, b;
  if (x.next == null)
    x.next = new Node(v, null);
  else if (a == b)
    x.next = new Node(v, x.next);
  else
    rand_insert(x.next, v);
}

void rand_insert2(Node2 x, int v) {
  int a, b;
  if (x.next == null)
    x.next = new Node2(v, x, null);
  else if (a == b) {
    x.next = new Node2(v, x, x.next);
    x.next.next.prev = x.next;
  } else
    rand_insert2(x.next, v);
}

void delete(Node x) {
  int a, b;
  if (x.next.next == null)
```

```
    x.next = null;
  else if (a == b)
    x.next = x.next.next;
  else
    delete(x.next);
}

void delete2(Node2 x) {
  int a, b;
  if (x.next.next == null)
    x.next = null;
  else if (a == b) {
    x.next = x.next.next;
    x.next.next.prev = x;
  } else
    delete2(x.next);
}

void append2(Node2 x, Node2 y) {
  Node2 w = x.next;
  if (w == null) {
    x.next = y;
    y.prev = x;
  } else
    append2(w, y);
}

void travrs2(Node2 x) {
  if (x.prev != null)
```

```
    travrs2(x.prev);
  if (x.next != null)
    travrs2(x.next);
}

int count(Node2 x) {
  int l = 0, r = 0;
  if (x.prev != null)
    l = count(x.prev);
  if (x.next != null)
    r = count(x.next);
  return 1 + l + r;
}

int height(Node2 x) {
  int l = 0, r = 0;
  if (x.prev != null)
    l = height(x.prev);
  if (x.next != null)
    r = height(x.next);
  if (l >= r)
    return 1 + l;
  else
    return 1 + r;
}

void insert2(Node2 x, int v) {
  Node2 p = x.prev;
  Node2 q = x.next;
```

```
  int a, b;
  if (a == b) {
    if (p != null)
      insert2(p, v);
    else
      x.prev = new Node2(v, null, null);
  } else {
    if (q != null)
      insert2(q, v);
    else
      x.next = new Node2(v, null, null);
  }
}

void delete2(Node2 x) {
  Node2 p = x.prev;
  Node2 q = x.next;
  int a, b;
  if (a == b) {
    if (p.prev == null && p.next == null)
      x.prev = null;
    else
      delete2(p);
  } else {
    if (q.prev == null && q.next == null)
      x.next = null;
    else
      delete2(q);
  }
```

```
}

int search(Node2 x, int v) {
  Node2 p = x.prev;
  Node2 q = x.next;
  if (x.val == v)
    return 1;
  else if (v < x.val) {
    if (p != null)
      return search(p, v);
    else
      return 0;
  } else {
    if (q != null)
      return search(q, v);
    else
      return 0;
  }
}

void bst_ins(Node2 x, int v) {
  Node2 p = x.prev;
  Node2 q = x.next;
  if (v < x.val) {
    if (p == null)
      x.prev = new Node2(v, null, null);
    else
      bst_ins(p, v);
  } else {
```

```
    if (q == null)
      x.next = new Node2(v, null, null);
    else
      bst_ins(q, v);
  }
}

Node2 avl_ins(Node2 x, int v) {
  if (x == null)
    return new Node2(v, null, null);
  else if (v < x.val) {
    x.prev = avl_ins(x.prev, v);
    if (height(x.prev) - height(x.next) == 2) {
      if (height(x.prev.prev) > height(x.prev.next))
        x = rot_lft(x);
      else
        x = rot_2_lft(x);
    }
  } else {
    x.next = avl_ins(x.next, v);
    if (height(x.next) - height(x.prev) == 2) {
      if (height(x.next.next) > height(x.next.prev))
        x = rot_rit(x);
      else
        x = rot_2_rit(x);
    }
  }
  return x;
}
```

```
Node append3(Node x, Node y) {
  if (x == null) return y;
  else {
    x.next = append3(x.next, y);
    return x;
  }
}

Node flatten(Node2 x) {
  if (x == null)
    return null;
  else {
    Node r = flatten(x.prev);
    Node s = flatten(x.next);
    Node t = new Node(x.val, null);
    Node a = append3(r, t);
    Node b = append3(a, s);
    return b;
  }
}

Node merge_sort(Node x) {
  if (x == null){
    return x;
  } else if (x.next == null) {
    return x;
  } else {
    int half = length(x) / 2;
```

```
    Node ctr = split(x, half);
    // Same split as the loop in Figure 4.2
    Node l = merge_sort(x);
    Node r = merge_sort(ctr);
    return merge(l, r);
  }
}

Node merge(Node x, Node y) {
  if (x == null)
    return y;
  else if (y == null)
    return x;
  if (x.next == null)
    return insert(y, x);
    // Same insert as the one in Figure 4.1
  else if (y.next == null)
    return insert(x, y);
  else if (x.val <= y.val) {
    Node t1 = merge(x.next, y);
    x.next = t1;
    return x;
  } else {
    Node t1 = merge(x, y.next);
    y.next = t1;
    return y;
  }
}
```

```
Node2 create2(int n) {
  if (n == 0) return null;
  else {
    Node2 r = create2(n - 1);
    Node2 s = unknown(n, r);
    if (r != null) r.prev = s;
    return s;
  }
}

void sort_insert(Node x, int v) {
  if (x.next == null)
    x.next = new Node(v, null);
  else if (v <= x.next.val)
    unknown(x, v);
  else
    sort_insert(x.next, v);
}

void sort_insert2(Node2 x, int v) {
  if (x.next == null)
    x.next = new Node2(v, x, null);
  else if (v <= x.next.val)
    unknown(x, v);
  else
    sort_insert2(x.next, v);
}

void tail_insert(Node x, int v) {
```

```
  if (x.next == null)
    unknown(x, v);
  else
    tail_insert(x.next, v);
}

void rand_insert(Node x, int v) {
  int a, b;
  if (x.next == null)
    x.next = new Node(v, null);
  else if (a == b)
    unknown(x, v);
  else
    rand_insert(x.next, v);
}

void rand_insert2(Node2 x, int v) {
  int a, b;
  if (x.next == null)
    x.next = new Node2(v, x, null);
  else if (a == b)
    unknown(x, v);
  else
    rand_insert2(x.next, v);
}

void delete(Node x) {
  int a, b;
  if (x.next.next == null)
```

```
    x.next = null;
  else if (a == b)
    unknown(x);
  else
    delete(x.next);
}

void delete2(Node2 x) {
  int a, b;
  if (x.next.next == null)
    x.next = null;
  else if (a == b)
    unknown(x);
  else
    delete2(x.next);
}

void travrs3(Node2 x) {
  Node2 y, z;
  unknown(x; y, z);
  if (y != null)
    travrs3(y);
}

int count(Node2 x) {
  Node2 y, z;
  int l = 0, r = 0;
  unknown(x; y, z);
  if (y != null)
```

```
    l = count(y);
  if (z != null)
    r = count(z);
  return 1 + l + r;
}

int height(Node2 x) {
  Node2 y, z;
  int l = 0, r = 0;
  unknown(x; y, z);
  if (y != null)
    l = height(y);
  if (z != null)
    r = height(z);
  if (l >= r)
    return 1 + l;
  else
    return 1 + r;
}

void insert2(Node2 x, int v) {
  Node2 p = x.prev;
  Node2 q = x.next;
  int a, b;
  if (a == b) {
    if (p != null)
      insert2(p, v);
    else
      unknown(x, v);
```

```
  } else {
    if (q != null)
      insert2(q, v);
    else
      x.next = new Node2(v, null, null);
  }
}

void delete2(Node2 x) {
  Node2 p = x.prev;
  Node2 q = x.next;
  int a, b;
  if (a == b) {
    if (p.prev == null && p.next == null)
      unknown1(x);
    else
      delete2(p);
  } else {
    if (q.prev == null && q.next == null)
      unknown2(x);
    else
      delete2(q);
  }
}

int search(Node2 x, int v) {
  Node2 p, q;
  unknown(x; p, q);
  if (x.val == v)
```

```
    return 1;
  else if (v < x.val) {
    if (p != null)
      return search(p, v);
    else
      return 0;
  } else {
    if (q != null)
      return search(q, v);
    else
      return 0;
  }
}

void bst_insert(Node2 x, int v) {
  Node2 p = x.prev;
  Node2 q = x.next;
  if (v < x.val) {
    if (p == null)
      unknown1(x, v);
    else
      bst_insert(p, v);
  } else {
    if (q == null)
      unknown2(x, v);
    else
      bst_insert(q, v);
  }
}
```

```
Node2 avl_ins(Node2 x, int v) {
  if (x == null)
    return new Node2(v, null, null);
  else if (v < x.val) {
    x.prev = avl_ins(x.prev, v);
    if (unknown(x.prev) - height(x.next) == 2) {
      if (height(x.prev.prev) > height(x.prev.next))
        x = rot_lft(x);
      else
        x = rot_2_lft(x);
    }
  } else {
    x.next = avl_ins(x.next, v);
    if (height(x.next) - height(x.prev) == 2) {
      if (height(x.next.next) > height(x.next.prev))
        x = rot_rit(x);
      else
        x = rot_2_rit(x);
    }
  }
  return x;
}

Node merge_sort(Node x) {
  if (x == null){
    return x;
  } else if (x.next == null) {
    return x;
```

```
  } else {
    int half = length(x) / 2;
    Node ctr = split(x, half);
    Node l = merge_sort(x);
    Node r = merge_sort(ctr);
    return unknown(l, r);
  }
}
```